Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

CONTROL DEL BRAZO ROBOT IRB120 MEDIANTE EL DISPOSITIVO HÁPTICO PHANTOM

Autor: Samuel Pardo Alía
Tutor/es: Elena López Guillén

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Control del brazo robot IRB120 mediante el dispositivo háptico PHANTOM"

Samuel Pardo Alía

2016

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

GRADO EN INGENERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

"Control del brazo robot IRB120 mediante el dispositivo háptico PHANTOM"

Autor:	Samuel Pardo Alía
Universidad:	Universidad de Alcalá
País:	España
Profesor Tutor:	Elena López Guillén

Presidente: D. Rafael Barea Navarro

Vocal 1: D. Manuel Ureña Molina

Vocal 2: Dña. María Elena López Guillén

CALIFICACIÓN:.....

FECHA:.....

Contenido general

Resumen

El proyecto expuesto en el presente libro consiste en el control del brazo robot IRB120 a través de diferentes aplicaciones por medio de una interfaz gráfica creada con el software matemático MATLAB. Todo ello se llevará a cabo gracias a un socket de comunicación desarrollado en lenguaje RAPID que permite el envío simultáneo de varias posiciones al robot.

Entre esas aplicaciones destaca el empleo del dispositivo háptico Phantom Omni para la creación de dibujos o textos a mano alzada que puedan ser reproducidos por el robot en un papel mediante un rotulador.

Por otro lado, este trabajo va a incluir la mejora de las aplicaciones desarrolladas en el proyecto "Reproducción de imágenes y textos con el robotIRB120" [1] de Guillermo Patiño, cuyo principal inconveniente residía en el elevado tiempo de ejecución.

Palabras clave: Phantom, dispositivo háptico, IRB120, MATLAB, socket, interfaz.

Abstract

The exposed project in the current book consists on the control of IRB120 robot's arm through different applications by means of a graphical interface created with the mathematical software MATLAB. All of this will be carried out thanks to a communication's socket developed in RAPID language that allows the simultaneous sending of several positions to the robot.

Between these applications, the most important is the employment of the haptic device Phantom Omni for artistic creation of freehand drawings or texts that could be reproduced by the robot on a paper with a marker.

Besides, this work is going to include the developed application's improvement in the project "Reproducción de imágenes y textos con el robot IRB120" [1] of Guillermo Patiño, whose main disadvantage resided in the high time of execution.

Resumen extendido

La tecnología avanza a pasos agigantados y prueba de ello lo tenemos en nuestra vida cotidiana, ya que hemos experimentado en apenas una década el desarrollo de los *smartphones* y, actualmente, resultan imprescindibles para la mayor parte de las personas, ya sea por ocio o para realizar tareas que antes sólo estaban al alcance de los ordenadores.

Estos avances afectan a diferentes campos de la tecnología, entre los que se encuentra el área de los robots, los cuales permiten agilizar procesos en sistemas de producción industrial o realizar tareas peligrosas para el ser humano, entre otras muchas cosas.

Una de las aplicaciones donde tienen cabida los sistemas robóticos es en la pintura, donde podemos encontrar diferentes robots que son capaces de realizar un cuadro o boceto. En esta idea se basa el trabajo "Reproducción de imágenes y textos" [1] de Guillermo Patiño, Departamento de Electrónica de la Universidad de Alcalá.

En dicho trabajo se adquieren imágenes o se escriben textos a través del ordenador, y tras aplicar diferentes tratamientos a las imágenes, se consiguen reproducir con el brazo robot IRB120 por puntos y líneas. Sin embargo, el tiempo de ejecución es demasiado elevado, por lo que en este trabajo se busca mejorar este inconveniente, además de optimizar las aplicaciones todo lo posible para obtener resultados más eficientes.

En el ámbito del arte, además, podemos destacar la aparición de los dispositivos hápticos, que son sistemas táctiles 3D con realimentación de fuerzas que permiten el modelado digital de esculturas y diseño de productos. Sin embargo, debido a sus amplias

posibilidades sus aplicaciones no se limitan a esto, sino que también pueden ser utilizados en odontología y en cirugías, pues poseen una elevada precisión.

La importancia de estos dispositivos y la posibilidad de poder trabajar con uno de ellos, permite que pueda realizarse el presente proyecto. Mediante el dispositivo háptico Phantom Omni se trabajará en una comunicación con el brazo robot IRB120 a través de MATLAB mediante librerías de código, con el fin de que el usuario sea capaz de realizar dibujos o escribir textos virtuales y que puedan ser enviados al brazo robot para que logre reproducirlos sobre un papel real.

Para poder realizar el envío de información de las aplicaciones al robot real se va a implementar un *socket* de comunicación según el protocolo TCP/IP, que tomará como base el proyecto realizado por Azahara Corbacho "Desarrollo de una interfaz para el control del robot IRB120 desde Matlab" [2], pero que, a diferencia de éste, permitirá el envío simultáneo de varios puntos para la generación de trayectorias fluidas, con el objetivo de conseguir el ahorro de tiempo en la ejecución.

Todo el conjunto de aplicaciones será implementado mediante la creación de una interfaz gráfica con la ayuda de la herramienta GUIDE de MATLAB, que incluirá diferentes botones, gráficos y ventanas con el fin de conseguir una comunicación eficaz y sencilla.

Lista de figuras

Lista de tablas

1. INTRODUCCIÓN

Desde siempre el ser humano ha buscado mejorar su calidad de vida y facilitar los trabajos forzosos con el objetivo de reducir los esfuerzos físicos y aumentar su productividad.

El origen de la robótica se remonta a miles de años atrás con la invención de artefactos, creados con materiales rudimentarios, que empleaban la fuerza bruta sirviendo de herramienta a los hombres. Con el paso del tiempo, el ser humano fue mejorando sus técnicas y con el empleo de distintos dispositivos y mecanismos como poleas o palancas desarrollaron los primeros autómatas, cuya primera finalidad fue lúdica. Sin embargo, no fue hasta el siglo XVIII, con la Revolución Industrial, cuando se impulsó el desarrollo de las máquinas, principalmente en la industria textil.

Los primeros robots fueron desarrollados en la década de los 50 y desde entonces el concepto de robótica ha evolucionado desde repetir una terea secuencialmente a poder disponer de cierta autonomía en sus acciones, llegando a simular el comportamiento de animales o incluso del propio ser humano.

Este desarrollo ha hecho que, en apenas 30 años, la robótica haya pasado de ser una ficción plasmada en obras literarias a convertirse en una realidad imprescindible en nuestro día a día.

1.1. Contexto

La robótica tiene un carácter interdisciplinario que aúna importantes áreas de la ingeniería como son la mecánica, la electrónica, la informática, la inteligencia artificial y la ingeniería de control. Es por ello que esta rama de la ingeniería supone un conjunto de conocimientos fundamental en la formación del ingeniero y es incluida en sus planes de estudios.

De este modo, en el *Grado en Ingeniería Electrónica y Automática Industrial* se incluye la asignatura de *Sistemas Robotizados* para dar formación a los futuros ingenieros en el mundo de la robótica. Los conocimientos impartidos abarcan desde el estudio de los brazos robots industriales, hasta el diseño de sistemas de control y programación de aplicaciones de automatización [3].

Además, cabe destacar que el Departamento de Electrónica de la Universidad de Alcalá dispone del brazo robótico IRB120 de ABB, lo que aumenta las posibilidades de aprendizaje del alumnado. Se permite así un contacto físico con el robot y no sólo mediante un simulador, facilitando una experiencia más interactiva y posibilitando el desarrollo de proyectos centrados en este brazo robótico. Por este motivo, en los últimos años se han desarrollado este tipo de proyectos, que son la base para el que se expondrá en el presente libro.

La idea principal tanto de este proyecto como de los que se han tomado como referencia, consiste en establecer una comunicación entre el brazo robot y un programa externo, que en este caso será MATLAB, con el objetivo de desarrollar una aplicación que será controlada mediante una interfaz:

- Azahara Gutiérrez Corbacho en su proyecto "Desarrollo de una interfaz para el control del robot IRB120 desde MATLAB" [2] establecía una comunicación (*socket*) entre el robot y el software matemático MATLAB a través de protocolos de Internet (TCP/IP). Esta comunicación sigue la estructura servidor-cliente en la que el bloque del servidor parte de un programa previo realizado en RAPID (lenguaje de programación del robot) por Marek Jerzy Frydrysiak llamado "Socket Based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station" [4].

- Basándose en los trabajos anteriores, Guillermo Patiño González en su proyecto "Reproducción de imágenes y textos con el robot IRB120" [1] empleaba el socket de comunicación entre MATLAB y el brazo robot con el objetivo de que el robot dibujase imágenes cargadas por el usuario o detectadas a través de la cámara, y escribiera textos reconocidos en una imagen o introducidos vía teclado.

El trabajo expuesto en este libro consistirá por una parte en mejorar el presentado en [1], que se basa en el socket de comunicación creado en [2] y [4] y que sólo permitía el envío de un dato de posición del robot cada vez, lo que daba un resultado poco eficiente; y, por otro lado, incluir en la comunicación el dispositivo háptico PHANTOM Omni ® para que el brazo robot IRB120 pueda reproducir los trazos realizados a mano alzada con el *lápiz* del dispositivo.

1.2. Objetivos

El objetivo inicial surge de la posibilidad de trabajar con un dispositivo háptico Phantom y ahondar en sus principales características con la finalidad de establecer una comunicación eficaz con el brazo robot IRB120 que permita reproducir con exactitud los movimientos realizados.

En paralelo, surge la posibilidad de realizar una mejora del proyecto "Reproducción de imágenes y textos con el robot IRB120" [1] realizado por Guillermo Patiño, quien desarrollaba una serie de aplicaciones que permitían que el brazo robot pudiera escribir textos y dibujar imágenes por puntos o líneas, o mediante una aplicación similar a Paint. Este proyecto tiene el inconveniente de que la transmisión de información entre la aplicación y el robot es muy lenta, por lo que la reproducción se demora en exceso.

Uniendo ambas ideas, se plantea un proyecto en el que se va a desarrollar una comunicación más fluida entre las aplicaciones y el robot, que además será empleada para la creación de una aplicación con el dispositivo Phantom cuya temática se basará en la reproducción de trazos realizados a mano alzada sobre un papel a través de un rotulador colocado en el efector final del brazo robot.

Todo ello se implementará mediante una interfaz gráfica desarrollada con la herramienta GUIDE de MATLAB, que permite una sencilla comunicación entre el usuario y el robot.

Por tanto, podemos resumir los objetivos del proyecto en los siguientes puntos:

- Establecer una comunicación fluida entre el entorno de creación de las aplicaciones y el entorno de ejecución del brazo robot.
- Mejorar las aplicaciones desarrolladas en el proyecto [1] incorporando la comunicación mencionada en el punto anterior.
- Desarrollar una aplicación con el dispositivo háptico Phantom que permita la reproducción de textos o imágenes dibujadas a mano alzada por el usuario.

1.3. Estructura de la memoria

En este apartado se realizará un breve resumen de cómo se encuentra organizada la memoria explicando a grandes rasgos lo que el lector encontrará en cada capítulo.

- **Capítulo 1. Introducción.** Este es el capítulo en el que nos encontramos, donde se presenta una breve introducción acerca de la importancia del desarrollo de la robótica y el contexto en que se sitúa el presente proyecto. Además, se exponen los antecedentes y objetivos que se van a tomar como base para su realización.

- **Capítulo 2. Herramientas Empleadas.** En este capítulo se aborda una descripción detallada sobre los elementos de hardware (brazo robótico IRB120, dispositivo háptico Phantom Omni) y software (MATLAB, RobotStudio, paquetes de librerías, protocolo de comunicación) que son necesarios para el desarrollo del proyecto.

- **Capítulo 3. Desarrollo de la Comunicación TCP/IP.** En este capítulo comienza el desarrollo del trabajo. Primero se expondrá el funcionamiento de los elementos que se toman como base de otros proyectos para después desarrollar el código implementado en la creación del nuevo datagrama de comunicación.

- **Capítulo 4. Aplicación de Dibujo con Phantom Omni.** Este es el capítulo principal del proyecto, donde se desarrollan los apartados primordiales para la calibración, adquisición y almacenamiento de los datos, y el posterior envío de la información al robot.

- **Capítulo 5. Mejora en Reproducción de Imágenes y Textos.** En este capítulo se realizará una breve introducción sobre la estructura del programa implementado en [3] para las diferentes aplicaciones, y posteriormente se desarrollarán los cambios introducidos para su mejora.

- **Capítulo 6. Desarrollo de la Interfaz Gráfica.** En este capítulo se expondrán las modificaciones introducidas sobre la interfaz implementada en [1] para la mejora de sus aplicaciones, y se mostrará la interfaz creada para la aplicación del Phantom Omni.

- **Capítulo 7. Conclusiones y Trabajos Futuros.** Como su propio nombre indica se mostrarán las conclusiones extraídas de la realización del proyecto, y se propondrán posibles trabajos atendiendo a las posibilidades de los elementos utilizados.

- **Capítulo 8. Planos.** En este capítulo se muestra parte del código desarrollado para la implementación de las aplicaciones.

- **Capítulo 9. Pliego de Condiciones.** Este capítulo expone las características de las herramientas utilizadas.

- **Capítulo 10. Presupuesto.** En este capítulo se estima el presupuesto necesario para la realización del proyecto.

- **Capítulo 11. Manual de Usuario.** En este capítulo se explicará detalladamente la utilización de cada una de las aplicaciones mediante la interfaz de usuario.

- **Capítulo 12. Bibliografía.** El último capítulo recoge cada una de las referencias a otros trabajos, libros o manuales que son utilizadas en el proyecto.

2. HERRAMIENTAS EMPLEADAS

En este apartado se expondrán de forma resumida las herramientas hardware y software necesarias para la realización de este proyecto.

2.1. Robot Industrial ABB

ABB (Asea Brown Boveri) es una corporación multinacional que desarrolla actividades industriales relacionadas con tecnologías de generación de energía eléctrica y de automatización industrial. Esta compañía se origina con la unión de ASEA y Brown, Boveri & Cie (BBC) en 1988, con sede central en Zúrich, Suiza.

ASEA surgió como una compañía eléctrica sueca que evolucionó hasta convertirse en un fabricante de electrónica con tecnología de alta precisión y se expandió en el área de la robótica. Por su parte, BBC nació como fabricante de componentes eléctricos para los ferrocarriles en Suiza y terminó dividida en filiales por toda Europa.

ABB se divide en 5 sectores de negocio: Productos de Potencia, que a su vez se subdivide en Productos para Alta Tensión, para Media Tensión y Transformadores; Sistemas de Potencia, que se subdivide en Sistemas de Red, Subestaciones, Manejo de Redes y Generación Eléctrica; Productos de Automatización, Automatización de Procesos y Robótica.

En el sector de la Robótica, ABB es considerado el mayor proveedor mundial de robots industriales (con más de 250.000 robots instalados), sistemas de producción modular para trabajos específicos y servicios. Su amplia gama de robots tiene como objetivo ayudar a los fabricantes a mejorar la productividad y calidad del producto manteniendo la seguridad del trabajador.

FIGURA 2.1 Ejemplos de robots ABB

2.1.1. Brazo robótico IRB120

Dentro del amplio catálogo de robots de ABB encontramos el IRB120 [5] como el robot que soporta menos carga (3 kg, que pueden ser 4 kg en posición vertical de la muñeca) y que tiene menor alcance (580 mm), lo que le convierte en el robot industrial multiusos más pequeño (tan sólo pesa 25 kg).

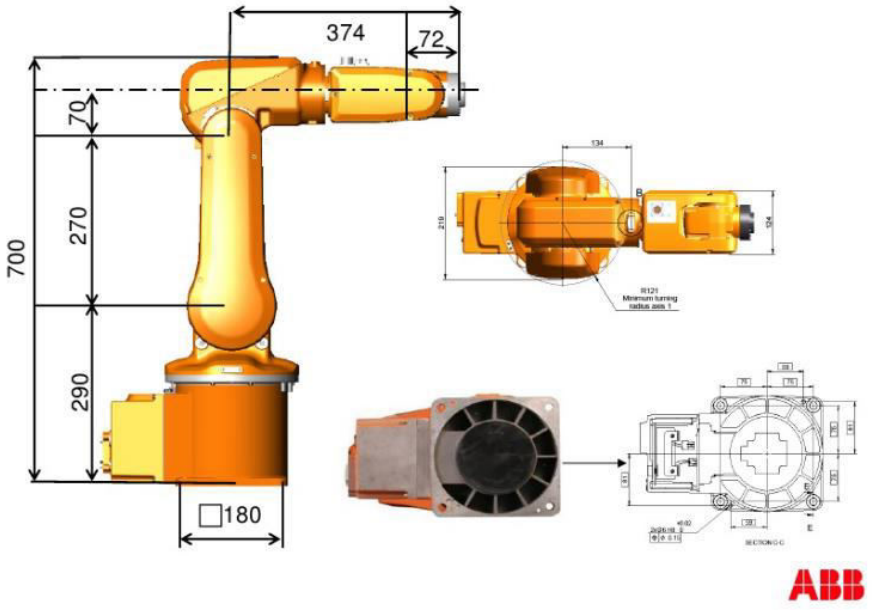


FIGURA 2.2 Robot IRB120

Las principales ventajas de este robot son su gran capacidad de producción con un bajo coste, su flexibilidad en operaciones frente a otras soluciones automatizadas más complejas y su fácil integración, ya que puede montarse en cualquier ángulo sin ninguna restricción como podemos observar en la siguiente imagen.

FIGURA 2.3 Diferentes montajes del IRB120

El robot IRB120 cuenta con 6 ejes, lo que le permite realizar 6 movimientos independientes respecto a su sistema de referencia para posicionar (3 primeras articulaciones) y orientar (3 últimas articulaciones, *muñeca*) su efector final. Podemos observar el movimiento que describe cada eje en la imagen. y los valores máximos de giro y velocidad de cada uno en la tabla.

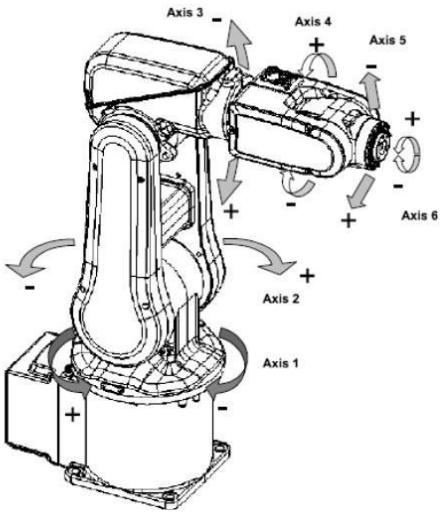


FIGURA 2.4 Ejes del robot IRB120

Ejes	Movimiento	Área Trabajo	Velocidad Máxima
Eje 1	Rotacional	+165º a -165º	250º/s
Eje 2	Angular	+110º a -110º	250º/s
Eje 3	Angular	+70º a -110º	250º/s
Eje 4	Rotacional	+160º a -160º	320º/s
Eje 5	Angular	+120º a -120º	320º/s
Eje 6	Rotacional	+400º a -400º	420º/s

TABLA 2.1 Movimiento de los ejes

Las delimitaciones del movimiento de cada uno de los ejes van a establecer el área de trabajo del robot. En la siguiente figura observamos el área de trabajo considerando como centro la muñeca.

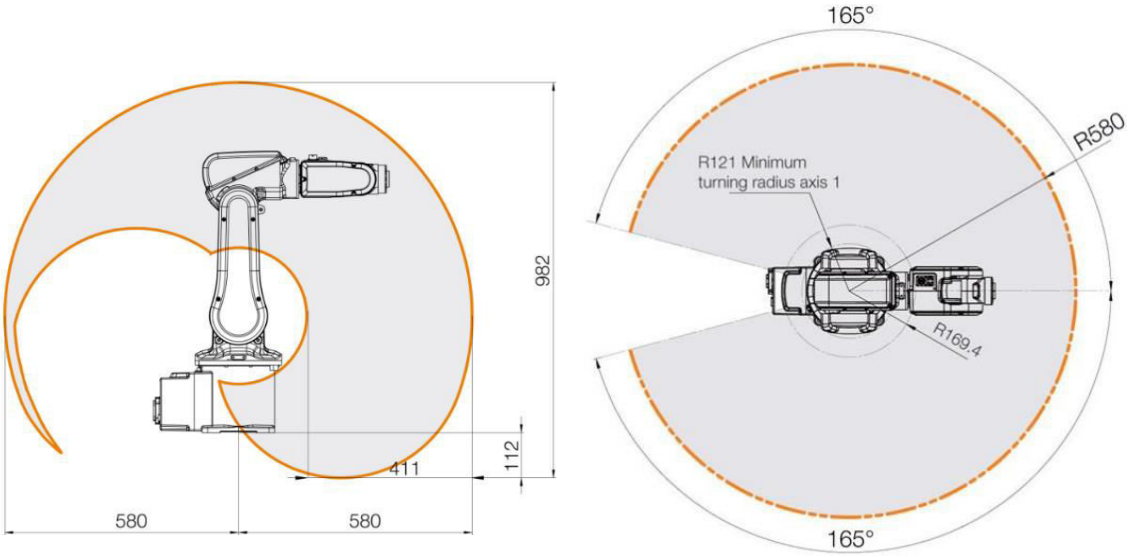


FIGURA 2.5 Área de trabajo de la muñeca del IRB120

2.1.2. Software RobotStudio

RobotStudio es un software de simulación y programación offline de ABB que permite crear, programar y simular células y estaciones de robots reales en un ordenador sin necesidad de disponer del robot real, lo que permite la reducción de riesgos, un arranque más rápido, transiciones más cortas y un aumento de la productividad.

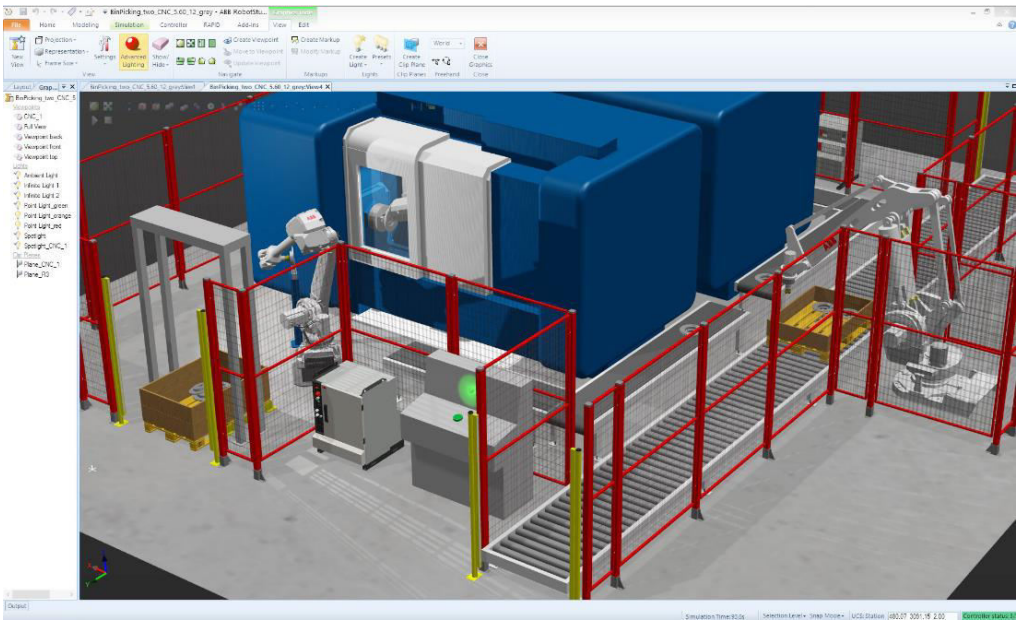


FIGURA 2.6 Ejemplo de estación en RobotStudio

RobotStudio es una copia exacta del software real que hace funcionar los robots de ABB, lo que permite simulaciones muy realistas permitiendo importar las configuraciones de los robots reales como podemos observar en la figura anterior.

Algunas de las características que podemos destacar de este simulador son:

- Importación de geometrías CAD y modelos 3D.
- Fácil diseño y creación de células robóticas.
- Programación y simulación cinemática de las estaciones con detección de colisiones del robot con su entorno.
- Exportación de las estaciones simuladas a las estaciones reales.

2.1.3. Lenguaje RAPID

El lenguaje RAPID (*Robotics Application Programming Interactive Dialogue*) es un lenguaje de programación de alto nivel desarrollado por la empresa ABB para el control de sus robots industriales. Este lenguaje nos permitirá el desarrollo del código correspondiente al socket de conexión entre el robot y MATLAB.

Una aplicación desarrollada en lenguaje RAPID está formada por un programa y una serie de módulos del sistema que permiten el control del robot como podemos ver en la siguiente figura.

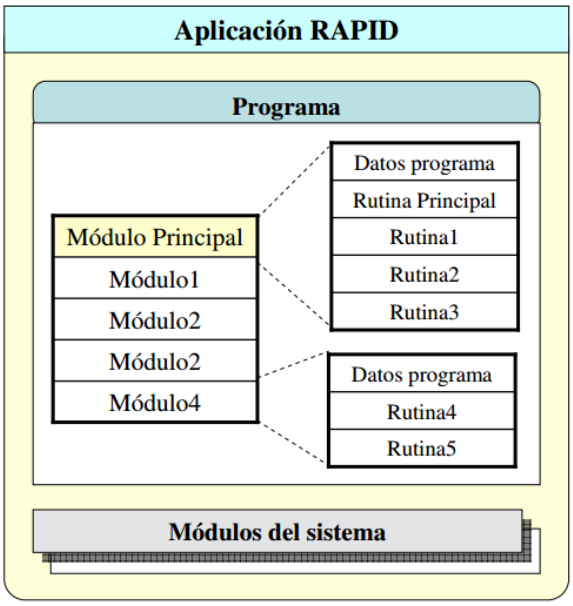


FIGURA 2.7 Estructura del lenguaje RAPID

Cada módulo del programa consta de una secuencia de instrucciones donde se distinguen tres partes:

- Rutina principal (*main*): inicio de la ejecución
- Subrutinas: dividen al programa principal en diferentes módulos.
- Datos del programa: declaraciones de variables

En la siguiente figura podemos observar la estructura de un programa.

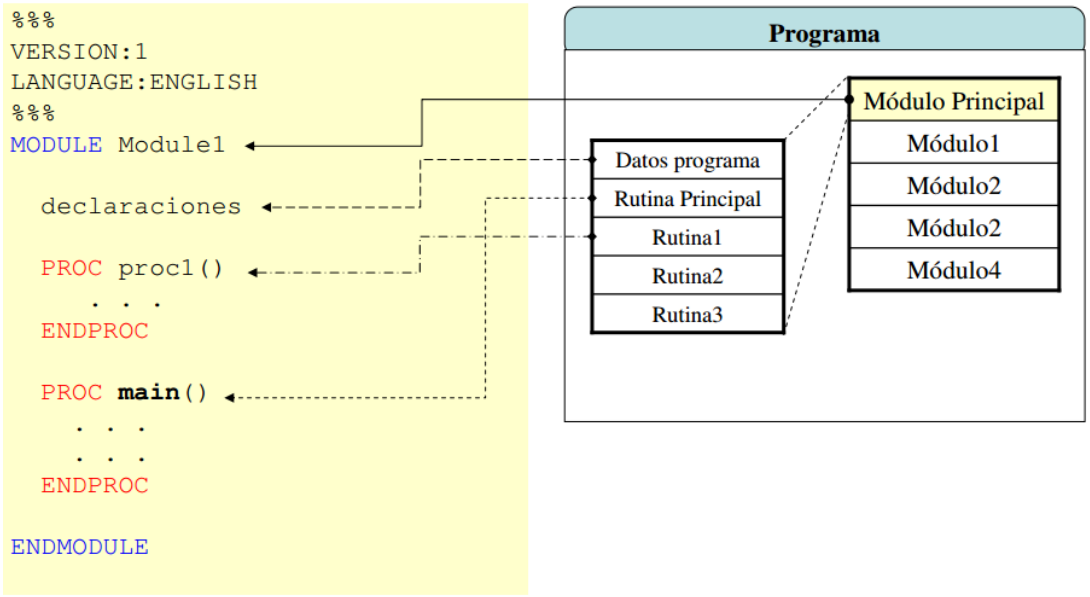


FIGURA 2.8 Estructura del programa RAPID

En las rutinas de un programa podemos encontrar:

- Procedimientos (*PROC*): son empleados como subrutinas
- Funciones (*FUNC*): devuelven un valor de un tipo específico y se utilizan como argumento de una instrucción.
- Interrupciones (*TRAP*).

Los datos del programa son la información que se manipula para poder desarrollar el algoritmo, de ahí que sean de vital importancia. En RAPID suelen denominarse mediante tres identificadores:

- Primer identificador: alcance
    - *GLOBAL*: cualquier módulo del programa tiene acceso a los datos.
    - *LOCAL*: sólo el módulo en el que se encuentra declarado el dato puede acceder a él.

- Segundo identificador: tipo de dato
  - *CONST*: datos con un valor fijo que no puede modificarse.
  - *VAR*: datos que pueden modificarse durante la ejecución del programa.
  - *PERS*: datos que cambian su valor de inicialización al modificarse durante la ejecución del programa.
- Tercer identificador: memoria de almacenamiento
  - Datos atómicos: unidad mínima de dato. Podemos encontrar *num*, *bool, string.*
  - Registros: permiten guardar más de un dato. Podemos encontrar en el manual de RAPID [6] un gran número de registros predefinidos.

Además de estos parámetros, el lenguaje RAPID incluye expresiones aritméticas y lógicas, manejo automático de errores y un conjunto de instrucciones.

2.1.4. Controlador IRC5 ABB y FlexPendant

El controlador IRC5 contiene los elementos y funciones necesarias para mover y controlar el robot manipulador. Está formado por un módulo de control, que incluye los elementos electrónicos de control, las tarjetas de E/S y la unidad de almacenamiento; y un módulo de accionamiento, que incorpora los elementos electrónicos de alimentación para los motores.

En función del armario, podemos encontrar diferentes variantes (*Dual, Single* y *Compact*) y en nuestro caso, dispondremos de la versión *Compact* que ofrece las capacidades de la variante *Single* en un formato reducido (310x449x442 mm y 30kg). Proporcionando así ahorro de espacio y una fácil puesta en marcha a través de la alimentación por una entrada monofásica.

FIGURA 2.9 Controlador IRC5 Compact de ABB

El IRC5 permite controlar los accionamientos de los 6 ejes del brazo robot, más tres accionamientos para ejes externos en función del tipo de robot. Desde un mismo módulo de control pueden dirigirse varios robots, pero debe existir un módulo de accionamiento por cada uno.

El FlexPendant es la unidad de programación que permite mover, calibrar y programar el sistema robot. Se presenta como una consola con pantalla táctil y un *joystick* 3D que concede una interacción más intuitiva. Además, cuenta con soporte para memoria USB, lo que permite cargar nuestros programas de RobotStudio fácilmente en el controlador del robot, conectado a este dispositivo mediante un cable integrado.

FIGURA 2.10 FlexPendant ABB

Para trabajar con el robot primero cargaremos el programa implementado en RAPID en el FlexPendant, desde donde también podremos editarlo. A continuación, arrancaremos el IRC5 y pulsaremos el botón de *play* en el FlexPendant para iniciar el programa. Cabe señalar que el FlexPendant dispone de un pulsador que debe mantenerse presionado durante la ejecución del movimiento del robot como medida de seguridad, aunque dispone también de un modo automático. En nuestra aplicación, el FlexPendant únicamente tendrá la función de intermediario entre nosotros y el robot, pero sin realizar ningún tipo de control, ya que esto lo implementamos desde un programa externo (MATLAB).

2.2. Sensable Technologies

Sensable Technologies Inc. es una compañía adquirida por *Geomagic* en 2012 que ofrece dispositivos hápticos[1] de realimentación de fuerza y soluciones software de modelado 3D.

Sus productos incluyen sistemas de modelado *FreeForm* con sistemas táctiles 3D para el modelado dental, diseño de productos, y modelado de esculturas para la creación de contenido digital y bellas artes.

Estos dispositivos permiten crear entornos virtuales de escultura donde los diseñadores pueden interactuar y sentir la forma en 3D de los objetos como si estuvieran diseñando un objeto real. Estos dispositivos hápticos pueden medir de forma precisa la posición espacial 3D (a lo largo de los ejes X, Y Z) y la orientación (giro, inclinación y dirección) del lápiz de mano, además, emplean motores para crear fuerzas de retorno en la mano del usuario que simulan el tacto cuando se interactúa con el modelo 3D en el espacio virtual.

Uno de los productos que oferta la compañía son los dispositivos hápticos Phantom que posibilitan una retroalimentación de fuerza, navegación en 3D, y trabajar en espacios virtuales. Además, ofrecen un kit de herramientas del software OpenHaptics para el desarrollo de aplicaciones táctiles.

FIGURA 2.11 Ejemplos de aplicaciones con dispositivos hápticos

[1] La percepción háptica es un sistema de percepción, integración y asimilación de sensaciones a partir del tacto activo que aúna la percepción táctil (estática, manos en reposo) y cinestésica (dinámica, movimiento voluntario de las manos) que permite percibir la forma y textura de los objetos.

2.2.1. Dispositivo háptico Phantom Omni ®

El dispositivo Phantom Omni (actualmente conocido como Geomagic Touch ™) [7] es el dispositivo háptico más económico disponible en el mercado. Construido con metal duradero y plásticos moldeados por inyección, permite la detección de la posición y orientación con 6 grados de libertad.

Además, admite un alto grado de flexibilidad gracias a su diseño portátil, su tamaño compacto, un lápiz extraíble y dos botones integrados. Emplea la interfaz FireWire®[2], que permite una rápida instalación y un fácil uso.

FIGURA 2.12 Dispositivo Phantom Omni y esquema de articulaciones

En la tabla inferior quedan reflejadas algunas de sus especificaciones técnicas:

Espacio de trabajo de fuerzas de retroalimentación	Ancho	De 6.4 a 160 mm
	Alto	De 4.8 a 120 mm
	Profundo	De 2.8 a 70 mm
Área de la base del dispositivo	Ancho	168 mm
	Profundo	203 mm
Resolución nominal de posición	>0.055 mm	
Fricción	<0.26N	
Máxima fuerza (brazos ortogonales)	3.3N	
Fuerza nominal (esfuerzo continuado)	0.88 N	
Dureza	Eje X	>1.26 N/mm
	Eje Y	>2.31 N/mm
	Eje Z	>1.02 N/mm
Masa aparente del lápiz	45 g	
Medida posición	Error de ±5% linealidad de potenciómetros	

TABLA 2.2 Especificaciones técnicas Phantom Omni

[2] La conexión FireWare® (IEEE 1394) es un tipo de conexión para distintas plataformas para la entrada y salida de datos serie a gran velocidad similar a la conexión USB.

En la siguiente figura simplificada podemos observar cómo se encuentran establecidos los ejes del Phantom, donde el giro del eje X corresponde a la *cintura* del dispositivo, el del eje Y sería el *hombro*, y el del eje Z se corresponde con el *codo*.

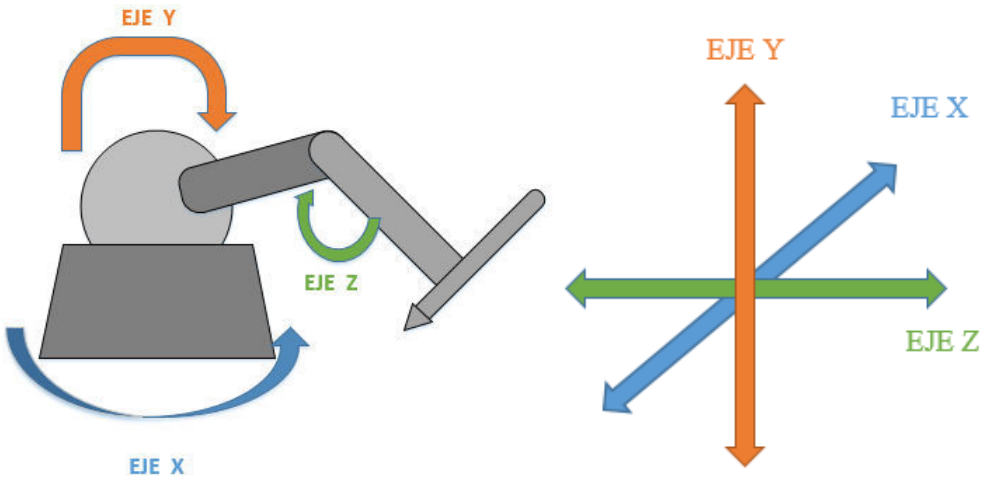


FIGURA 2.13 Disposición de los ejes en el Phantom Omni

2.2.2. Haptik Library

Haptik es una librería de código abierto desarrollada por investigadores del *Siena Robotics and Systems Lab.* [8] en Italia, que proporciona un *layer* (capa) de abstracción de hardware para el acceso a dispositivos hápticos de distintos fabricantes de una manera uniforme, permitiendo eliminar las dependencias de aplicaciones en configuraciones de APIs (Interfaz de Programación de Aplicaciones), hardware y drivers particulares.

Esta librería [9] no contiene elementos gráficos ni una enredada jerarquía de clases, sino que ha sido diseñada para ser fácil de usar incluso en aplicaciones complejas.

Puede integrarse fácilmente mediante programación con clases permitiendo un acceso basado en *callbacks* y puede usarse tanto en sistemas de coordenadas de *mano derecha* (OpenGL[3]) como en sistemas de *mano izquierda* (DirectX[4]).

[3] OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que producen gráficos 2D y 3D.
[4] DirectX es una colección de APIs desarrolladas para facilitar tareas multimedia, especialmente la programación de juegos y videos en la plataforma Microsoft Windows.

Haptik Library consiste en una recarga dinámica de *plugins*[5] de ejecución lo que hace que pueda ampliarse y personalizarse fácilmente, y permita eliminar la dependencia del tiempo de compilación en un conjunto particular de DLLs.

Por otra parte, desde las diferentes librerías existentes basadas en una arquitectura de componentes, se garantiza compatibilidad binaria de las aplicaciones compiladas por el cliente tanto en versiones anteriores como futuras de dispositivos hardware, drivers, plugins y de la propia librería, manteniendo al mismo tiempo el máximos rendimiento alcanzable directamente empleando SDKs nativas (OpenHaptics).

Además, estas librerías no sólo pueden utilizarse desde lenguaje C++, sino también en entornos de programación como MATLAB (donde estará desarrollado este proyecto) y Java.

- **Funciones:**

Las aplicaciones con MATLAB sólo tienen como restricción el acceso a *haptik_matlab.dll, haptikdevice_list* y la carpeta *@haptikdevice*. Haptik library tiene una API muy sencilla, y tan sólo tres líneas de código son necesarias para empezar a utilizar cualquier dispositivo. Las funciones principales [10] de esta librería son:

- *haptikdevice_list*: imprime el listado de los dispositivos hápticos conectados.
- *h=haptikdevice (id)*: carga el dispositivo con la id correspondiente del listado de conectados.
- *h=haptikdevice*: obtiene la información del dispositivo por defecto.
- *botón=read_button(h)*: lee el estado de los pulsadores del dispositivo. Si se pulsa el botón gris, aparecerá un 1; si se pulsa el botón blanco, un 2; y si se pulsan ambos botones un 3. Esta es una función que nos será de gran utilidad como veremos en apartados posteriores.
- *posición=read_position(h)*: lee la posición del dispositivo (valor de los ejes X, Y y Z) en milímetros.
- *[matriz, botón]=read(h)*: obtiene la posición y orientación del dispositivo en matrices 4x4 donde los elementos de la última fila corresponden a la posición. Además, indica el botón pulsado.

[5] Un *plugin* es una aplicación que se relaciona con otra para aportarle una función nueva y específica.

- *write (h, ff)*: envía fuerzas y torques al dispositivo mediante una matriz ff que puede ser 1x3, 3x1, 1x6, 6x1, 2x3, 3x2.
- *close(h)*: cierra el dispositivo.

2.3. MATLAB

MATLAB (*MATrix LABoratory*) es una herramienta de software matemático que tiene su propio lenguaje de programación (lenguaje M) y ofrece un IDE (Entorno de Desarrollo Integrado). Se encuentra disponible en las principales plataformas como Windows, MAC OS X o GNU/Linux y es utilizado ampliamente por ingenieros y científicos, debido a la naturalidad de su lenguaje basado en matrices, para expresar las matemáticas computacionales.

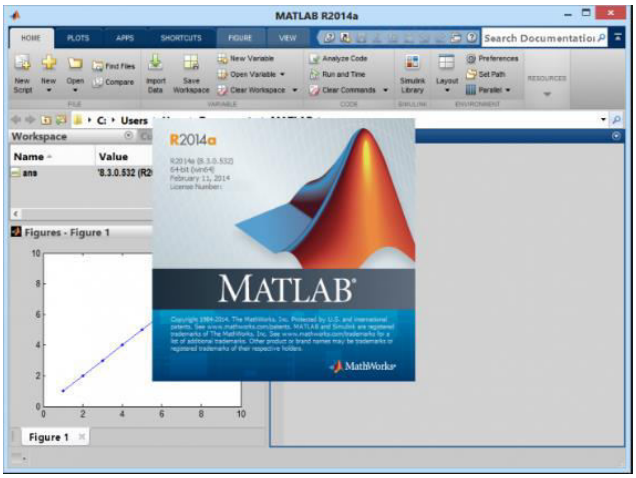


FIGURA 2.14 Interfaz MATLAB R2014a

Cuenta con infinidad de prestaciones: además de la manipulación de matrices, integra la representación de datos y funciones mediante gráficos, lo que facilita su análisis y permite su visualización; también admite el desarrollo de algoritmos y aplicaciones que pueden integrarse en otros lenguajes.

Presenta una amplia librería de *toolboxes* preinstaladas que hacen posible que este software pueda emplearse para numerosas aplicaciones desde el procesamiento de señales o imágenes hasta finanzas computacionales. Además, permite ampliar sus capacidades añadiendo nuevas librerías.

A todo esto, se suman dos herramientas preinstaladas:

- Simulink: plataforma que posibilita la simulación de sistemas dinámicos mediante una interfaz basada en bloques. Permite añadir paquetes de bloques que amplía sus posibilidades.
- GUIDE: editor de interfaces de usuario que logra el desarrollo de aplicaciones y que se empleará en el presente proyecto.

2.3.1. Toolboxes empleadas

Como se ha mencionado, las toolboxes [11] son una parte importante de MATLAB ya que incluyen funciones específicas que permiten desarrollar aplicaciones en diversos campos. En este proyecto se emplearán tanto toolboxes preinstaladas como descargadas adicionalmente. A continuación, se expondrán las que resultan de mayor interés:

- **Toolbox de Robótica (*Robotics Vision & Control Toolbox*):**

Toolbox desarrollada por Peter Corke [12] que ofrece funciones útiles para el estudio y simulación de brazos robóticos clásicos. Se basa en el método general de representación de la cinemática y dinámica de manipuladores con varias articulaciones.

Esta toolbox además proporciona funciones para el manejo y conversión entre diferentes tipos de datos como vectores, trasformaciones homogéneas y *cuaternios*, que permiten la representación tridimensional de la posición y orientación del robot.

Las posibilidades de esta librería hacen que sea esencial para poder trabajar en el presente proyecto en MATLAB con el IRB120.

- **Toolboxes para el tratamiento de imágenes:**

Estas toolboxes son necesarias para la reproducción de imágenes y textos desarrollada en [1] y permiten desde la adquisición de imágenes desde una cámara hasta su transformación aplicando técnicas de segmentación.

- Toolbox de Visión por Computadora (*Computer Vision System*): proporciona algoritmos, funciones y aplicaciones para el diseño y simulación de sistemas de visión por ordenador y de procesamiento de video. Permite la detección y seguimiento de objetos, y procesamiento de video. Para sistemas tridimensionales, la toolbox cuenta con calibración de cámara, reconstrucción 3D y procesamiento

de nubes de puntos. La característica que ha sido empleada en este proyecto es la que permite el reconocimiento de textos en una imagen. La principal función empleada es:

- *txt=ocr(I, roi)*: reconoce el texto de una imagen empleando un reconocimiento óptico de caracteres. El campo *I* hace referencia a la imagen, mientras que el campo *roi* contiene la región de interés especificada en matrices de 4 elementos.

- Toolbox de Procesamiento de Imágenes (*Image Processiong Toolbox*): incluye un conjunto de algoritmos, funciones y aplicaciones para el procesamiento, análisis y visualización de imágenes. Permite analizar imágenes, segmentarlas, mejorarlas mediante una reducción de ruido y realizar transformaciones, funciones que serán utilizadas en este trabajo. Las principales funciones utilizadas son:
  - *I=imread(filename)*: lee la imagen del archivo especificado por *filename*.
  - *imshow(I)*: muestra la imagen por pantalla.
  - *gray=rgb2gray(I)*: convierte la imagen *I* en una escala de grises eliminando la información de tono y saturación al tiempo que conserva la luminancia.
  - *im2bw (gray, level)*: convierte una imagen *gray* de escala de grises a formato binario reemplazando los pixeles de *gray* con luminancia mayor que *level* por un 1 (blanco) y sustituyendo el resto por un 0 (negro).
  - *SE=strel ('line', len, deg)*: crea un elemento de estructura linear que es simétrico respecto al centro de la región. El campo *deg* especifica el ángulo (en grados) de las líneas medidas en la dirección contraria a la de las agujas del reloj respecto al eje horizontal, mientras que el campo *len* es aproximadamente la distancia entre los centros de los elementos de estructuras en extremos opuestos de la línea.
  - *imdilate (IM, SE)*: dilata una imagen *IM* en escala de grises o en formato binario. El argumento *SE* es un objeto de un elemento de una estructura o un array de objetos devueltos por la función *strel*.
  - *imerode (IM, SE)*: *erosiona* una imagen *IM* en escala de grises o en formato binario. El argumento *SE* es un objeto de un elemento de una estructura o un array de objetos devueltos por la función *strel*.

- *tform=maketform ('projective', U, X)*: construye una estructura *TFORM* para una transformación proyectiva de 2 dimensiones que mapea cada fila de *U* para la fila correspondiente de *X*. Los argumentos *U* y *X* son de dimensiones 4x2 y definen las esquinas de entrada y salida de los cuadriláteros.
- *imtransform (A, tform, 'bicubic', 'udata', udata, 'vdata', vdata, 'size', size 'fill', fill)*: transforma la imagen *A* acorde a la trasformación espacial bidimensional definida por *tform* especificando la forma de interpolación que se va a usar (*'bicubic'*). Los campos *'udata'* y *'vdata'* son vectores de dos elementos que se combinan entre sí para especificar la localización espacial de la imagen *A* en el espacio bidimensional de entrada U-V. El argumento *'size'* especifica el número de filas y columnas de la imagen resultante de la transformación, mientras que el campo 'fill' se emplea para rellenar los pixeles de salida cuando la correspondiente zona trasformada de la imagen de entrada está completamente fuera de sus límites.

- **Toolbox de Control de Instrumentos (*Instrument Control Toolbox*):** permite la conexión de MATLAB con instrumentos como osciloscopios y generadores de señales. Se conecta a través de drivers de instrumentos como IVI y VXIplug&play o por medio de comando SCPI basados en textos a través de protocolos de comunicación comunes como GPIB, VISA, TCP/IP y UPD. Esta librería será empleada para la comunicación de MATLAB con el robot IRB120 (o el simulador RobotStudio) para enviar datos y leer los recibidos para su análisis. Entre las principales funciones que emplearemos encontramos:
  - *obj=tcpip (host, port, 'PropertyName', PropertyValue, 'NetworkRole', role)*: posibilita la creación de un objeto TCPIP. El campo *host* determina si el host (anfitrión)[6] es remoto (*'rhost'*) o local (*'localhost'*) con su correspondiente puerto (*rport*) de conexión. El campo *'PropertyName'* especifica un valor de propiedad, mientras que *'NetworkRole'* determina la función como cliente o servidor en la conexión.
  - *set (obj, 'PropertyName', PropertyValue)*: configura o muestra las propiedades del objeto creado.

[6] El término *host* se utiliza en informática para referirse a ordenadores conectados a la red.

- *fopen(obj)*: conecta *obj* (objeto interfaz) con el instrumento (que en nuestro caso será el robot).
- *A=fread (obj, size,'precision')*: lee datos binarios de un instrumento. El campo *size* determina el número de valores que se van a leer, mientras que *'precision'* establece el número de bits leídos de cada valor y la interpretación de los bits como valor entero, carácter o punto flotante.
- *fwrite (obj, A, 'precision', 'mode')*: escribe un dato binario (A) en el instrumento. El campo *'precision'* tiene la misma función que en el caso anterior, pero determinando los bits que van a escribirse; el campo '*mode'* especifica si los datos se escriben de manera síncrona o asíncrona.

2.3.2. Clases en MATLAB

La programación con objetos es un enfoque de programación formal que combina datos y métodos (acciones) en estructuras lógicas (objetos) que aumenta la capacidad para administrar un software complejo, que es muy importante al desarrollar aplicaciones y estructuras de datos grandes. La creación de clases nos permitirá en el presente proyecto la comunicación con el robot desde cualquier archivo de MATLAB.

La utilización de objetos en MATLAB sirve para realizar aplicaciones complejas con mayor rapidez que otros lenguajes como C++, C# y Java. La definición de clases permite la reutilización de código, el encapsulamiento, la herencia y el comportamiento de referencia sin prestar atención a tareas de bajo nivel como otros lenguajes.

Para programar con objetos en MATLAB es necesario utilizar:

- **Archivos de definición de clases**:

Permiten la definición de propiedades, métodos y eventos.

- Propiedades: posibilita el almacenamiento de datos que después empleará la clase.
- Métodos: funciones que implementan operaciones dentro de la clase.
- Eventos: mensajes definidos por las clases que se ejecutan cuando se produce una acción.

- **Clases con comportamiento de referencia:**

Ayudan a la creación de estructuras de datos como listas vinculadas.

- **Eventos y oyentes:**

Permiten monitorizar las acciones y los cambios de las propiedades de los objetos.

- Oyentes: objetos que responden ante un evento ejecutando una función de devolución de llamada.

En el presente proyecto no se va a crear una clase, sino que se modificará la realizada por Azahara Gutiérrez Corbacho en su proyecto [2] para añadir métodos necesarios para la implementación de este trabajo. Por este motivo, no se entrará en más detalle en este apartado.

2.3.3. Interfaces gráficas GUI

Como ya se ha mencionado, una de las posibilidades que incluye MATLAB es la creación de GUIs (Interfaces Gráficas de Usuario) que permiten un control sencillo de las aplicaciones software mediante menús, barras de herramientas, botones y controles deslizantes, facilitando la manera en la que el usuario interactúa con el programa, sin necesidad de estar escribiendo líneas de código.

Para poder crear estas interfaces MATLAB dispone de GUIDE, que es un entorno de programación visual para realizar y ejecutar GUIs. GUIDE genera de manera automática el código de MATLAB para construir la interfaz, que puede ser modificado para cambiar cualquier funcionalidad.

Para ejecutar GUIDE podemos teclear en la ventana de comandos *guide* o bien en *Home->New->GUI* y se abrirá una ventana donde se nos permitirá abrir una GUI existente, comenzar a trabajar desde un ejemplo o crear una desde el principio (por defecto).

Una vez elijamos una de las opciones se abrirá un archivo *.fig* donde podremos empezar a trabajar, que se muestra en la siguiente imagen:

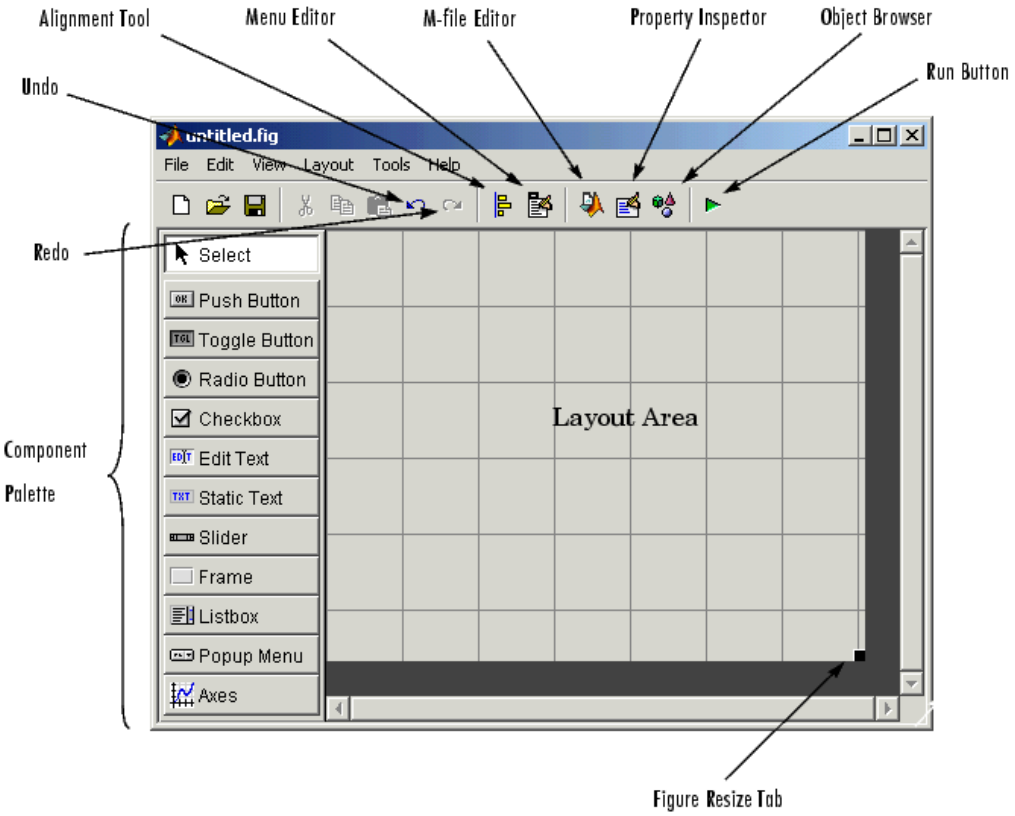


FIGURA 2.15 Zona de trabajo GUIDE

El área de trabajo de GUIDE puede dividirse en:

- **Barras de herramientas:**

Barra de Herramientas	
Alignment Tool	Ajusta la posición de un objeto respecto a otro, proporcionando una alineación tanto vertical como horizontal.
Menu Editor	Permite diseñar barras de herramientas en los GUI (*Menu Bar*) y los menús que aparecen en los objetos (*Context Menus).*
M-file Editor	Permite editar el código creado por la interfaz en un archivo.*m*.
Property Inspector	Permite asignar y modificar las propiedades de cada objeto.
Object Browser	Muestra los objetos que se encuentran en la figura en forma de árbol y permite su selección.
Run Button	Permite crear el GUI y ejecutarlo.

TABLA 2.3 Barra herramientas GUIDE

- **Paleta de componentes:**

Aquí se encuentran los componentes que se pueden insertar al área de trabajo y que reciben el nombre de *uicontrols.*

Paleta de componentes	
Push Button	Invoca un evento al ser pulsado.
Toggle Button	Cada vez que se pulsa conmuta de estado entre on/off.
Radio Button	Indica una opción que puede seleccionarse o no.
Check Box	Indica el estado de una opción mediante un *tick.*
Edit Text	Permite introducir un recuadro con texto editable.
Static Text	Es igual que el control anterior, pero en este caso se mostrará siempre el mismo texto sin opción de modificación.
Slider	Se utiliza para representar un rango de valores entre un máximo y un mínimo.
List Box	Genera una lista deslizable con varias opciones para elegir.
Pop-up Menu	Permite crear un menú desplegable con diferentes opciones.
Axes	Permite insertar ejes y figuras en la GUI.

TABLA 2.4 Controles GUIDE

Cada uno de los controles tiene un conjunto de opciones a las que se pueden acceder mediante *click derecho.* Una de las más importantes es *View Callbacks,* que al ejecutarla abre el archivo *.m* situándose en la función que se ejecutará cuando se realice una acción sobre el elemento en cuestión.

Todos los valores de los elementos y variables del GUI se almacenan en una estructura a la que se accede mediante el identificador *handles.* Para garantizar que cualquier cambio o asignación de propiedades o variables queda guardado se utiliza la sentencia *guidata:*

```
handles.output = hObject;
            ...
guidata(hObject, handles);
```

Para asignar u obtener valores de los componentes, se utilizan las sentencias *get* y *set:*

- *set (id, 'propiedad', 'valor')*: establece un nuevo *'valor'* para la *'propiedad'* del objeto con identificador *id.*
- *get (id, 'propiedad')*: permite obtener el valor de la *'propiedad'* seleccionada.

2.4. Socket TCP/IP

Un *socket* es un método de comunicación entre distintos procesos que se suele utilizar para enlazar diferentes dispositivos que están conectados a la misma red con el objetivo de poder intercambiar flujos de datos entre ellos.

Los sockets son creados dentro de un dominio de comunicación que indica el formato de las direcciones que podrán tomar y los protocolos que soportarán. Existen diferentes dominios de comunicación, pero emplearemos el dominio *AF_INET (Adress Family INET)* que son sockets de Dominio de Internet, ya que son el tipo más genérico.

El funcionamiento de los sockets se puede simplificar en tres pasos:

- Primero se ejecuta el socket que se encargará de recibir los datos, que llamaremos *servidor*, y que se mantendrá a la espera.
- A continuación, se ejecuta otro socket que se encarga de enviar los datos al servidor, y que denominaremos *cliente*.
- Por último, el cliente realizará una petición al servidor, y este gestionará la respuesta, que será enviada al cliente, estableciéndose la conexión.

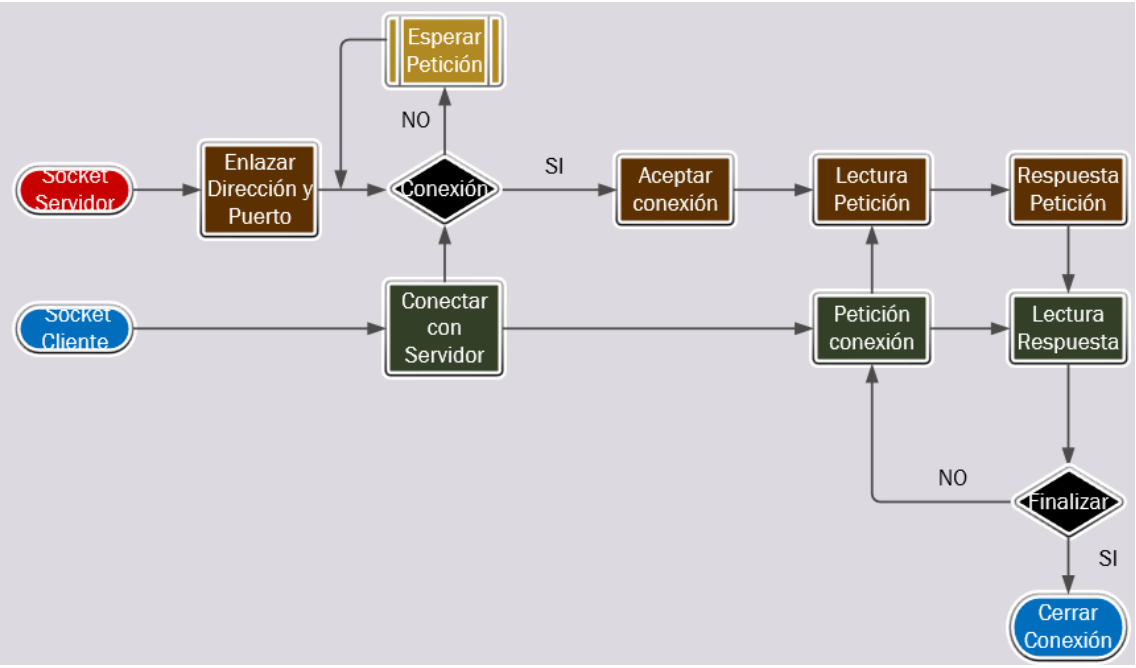


FIGURA 2.16 Esquema conexión sockets

De este modo, los sockets implementan una arquitectura cliente-servidor que permite que el programa servidor y cliente puedan leer y escribir información, produciéndose una transmisión bidireccional.

Para que puedan establecerse la comunicación son necesarios dos recursos:

- Direcciones de protocolo de red para identificar el ordenador de origen y el remoto.
- Números de puerto para identificar los programas dentro del ordenador.

2.4.1. Comunicación TCP/IP

TCP/IP (Protocolo de Control de Transmisión / Protocolo de Internet) es un conjunto de protocolos de red que se basa en Internet para la comunicación de datos entre diferentes máquinas.

Este modelo de comunicación se encuentra dividido en 4 *capas*, que se corresponden con módulos que realizan tareas específicas secuencialmente. Las capas de mayor a menor nivel son las siguientes:

- **Capa de aplicación**: incorpora aplicaciones de red estándar (Telnet, SMTP, FTP…)

Es la capa superior del protocolo TCP/IP y contiene las aplicaciones de red que permiten la comunicación con las capas inferiores. La mayoría de las aplicaciones son servicios de red o aplicaciones para proporcionar la interfaz con el sistema operativo.

- **Capa de transporte**: ofrece los datos de enrutamiento junto con los mecanismos para conocer el estado de la transmisión.

Posibilita que las aplicaciones que se ejecutan en equipos remotos puedan comunicarse entre sí. Para poder identificar las aplicaciones se utiliza un sistema de numeración mediante puertos.

Esta capa tiene dos tipos de protocolos que permiten que dos aplicaciones puedan intercambiar datos independientemente del tipo de red. Dependiendo del protocolo con el que se realice la conexión podemos encontrarnos con dos tipos de sockets:

Protocolo TCP	Protocolo UDP
Está orientado a la conexión	No está orientado a la conexión
Se utiliza para comunicar dos ordenadores a través de Internet mediante conexiones virtuales.	Se usa para el transporte de mensajes, pero al no estar basado en conexiones, tras el envío de datos termina la relación.
Es útil para aplicaciones que requieren alta seguridad en que todos los datos van a ser enviados y recibidos en el mismo orden.	No tiene un orden de envío y los paquetes de datos son independientes, por lo que no puede garantizar la transmisión de todos ellos.
El tiempo de transmisión es elevado ya que requiere 3 envíos para la conexión con el socket (petición de conexión, aceptación de conexión y respuesta a petición) y controla la recepción de todos los datos.	Es útil para aplicaciones que necesitan una transmisión rápida y efectiva con un elevado número de peticiones de pequeños datos, ya que no hace una verificación de errores de cada paquete.

FIGURA 2.17 Comparativa protocolos TCP y UDP de la comunicación TCP/IP

Tras conocer las características de cada protocolo, para realizar el socket de comunicación de nuestro programa, se optó por el protocolo TCP debido a que las aplicaciones requieren de cierta precisión por lo que la recepción de los puntos enviados en el orden de envío es de vital importancia, a pesar de que la velocidad de transmisión sea más lenta.

- **Capa de Internet:** se encarga de proporcionar el paquete de datos.

Es la capa con mayor relevancia ya que define los paquetes de datos y administra las direcciones IP. Esta capa consta de 5 protocolos:

- Protocolo IP: es uno de los protocolos de Internet más importantes porque permite el desarrollo y transporte de datagramas de IP.
- Protocolo ARP: permite que se conozca la dirección física de una tarjeta de interfaz de red correspondiente a una dirección IP.
- Protocolo ICMP: permite administrar la información relacionada con errores de los equipos de red.
- Protocolo RARP: posibilita que la estación de trabajo conozca su dirección IP desde una tabla de búsqueda entre las direcciones físicas y las direcciones IP ubicadas en la misma red de área local (LAN). Es muy poco utilizado, sólo se emplea en estaciones de trabajo sin discos duros.
- Protocolo IGMP: se utiliza para intercambiar información sobre el estado de pertenencia entre los enrutadores IP que admiten multidifusión (envío a múltiples redes) y los miembros de grupos de multidifusión.

- **Capa de acceso a la red:** especifica la forma en la que los datos deben *enrutarse* (conectarse a la red).

Es la capa de menor nivel y ofrece la posibilidad de acceder a cualquier red física, ya que contiene las especificaciones relacionadas con la transmisión de datos por una red de área local, conectada por línea telefónica o por fibra óptica.

3. DESARROLLO DE LA COMUNICACIÓN TCP/IP

En este capítulo se abordará la modificación realizada sobre el socket de comunicación para poder desarrollar las aplicaciones propuestas.

3.1. Arquitectura del sistema

La comunicación es la parte fundamental del sistema ya que es el canal de transmisión de los flujos de datos entre los diferentes dispositivos y plataformas. Para poder entender las partes que intervienen en la comunicación resulta útil desarrollar un esquema en el que se refleje la estructura general que va a seguir el sistema:

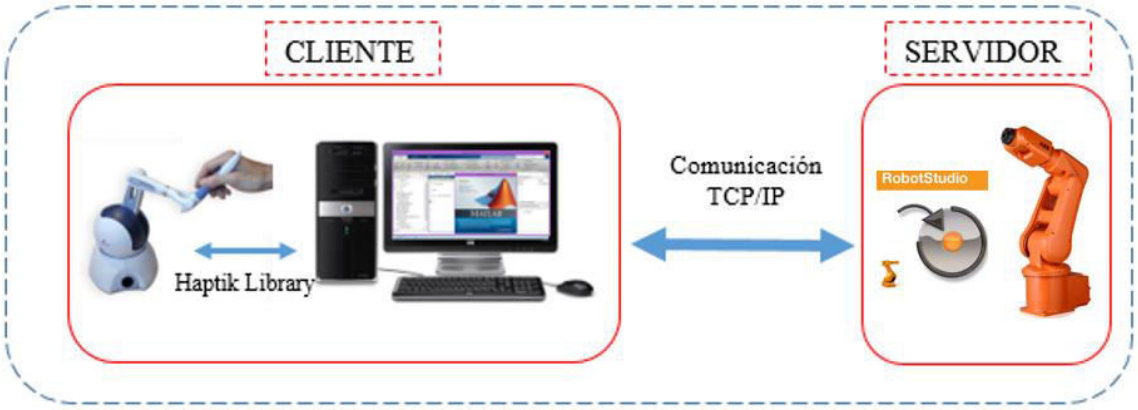


FIGURA 3.1 Esquema general del sistema

En el esquema anterior podemos comprobar que nuestro sistema engloba dos bloques fundamentales conectados mediante la comunicación TCP/IP.

En el bloque de la izquierda, el cliente, encontramos tanto el dispositivo Phantom Omni como el ordenador necesario para ejecutar el programa MATLAB. La conexión entre ambos, además del cable que se conecta al puerto FireWire ® del ordenador, se produce a través de los drivers y librerías instaladas del dispositivo háptico. Del mismo modo la aplicación de reproducción de imágenes y textos [1] permitía la adquisición de imágenes mediante una cámara conectada al ordenador o la propia webcam integrada, que podríamos considerar un periférico más. Este bloque será el encargado de establecer la comunicación con el bloque servidor y enviarle la información necesaria para que el robot ejecute la aplicación.

Por otro lado, el bloque de la derecha, el servidor, se ha representado mediante la imagen del robot IRB120 de ABB y su correspondiente software de simulación RobotStudio, que desempeña una labor importante a la hora de depurar el programa RAPID necesario para la conexión, por tanto, el cliente va a poder comunicarse por ambas vías. Los resultados en ambas plataformas deben ser idénticos si se ha establecido la conexión de forma correcta. Este bloque estará a la escucha del bloque servidor, manteniéndose a la espera hasta que se establezca la conexión.

3.2. Implementación del servidor

Como ya se ha mencionado, esta parte del proyecto toma como base el trabajo realizado por Marek Jerzy Frydrysiak [3] donde se desarrollaba un socket de comunicación entre un servidor (RAPID) y un cliente (lenguaje C), sobre el que debemos realizar modificaciones para poder desarrollar nuestra aplicación de forma correcta.

En primer lugar, detallaremos en el siguiente esquema el funcionamiento de este servidor, que se considera básico para el entendimiento del programa.

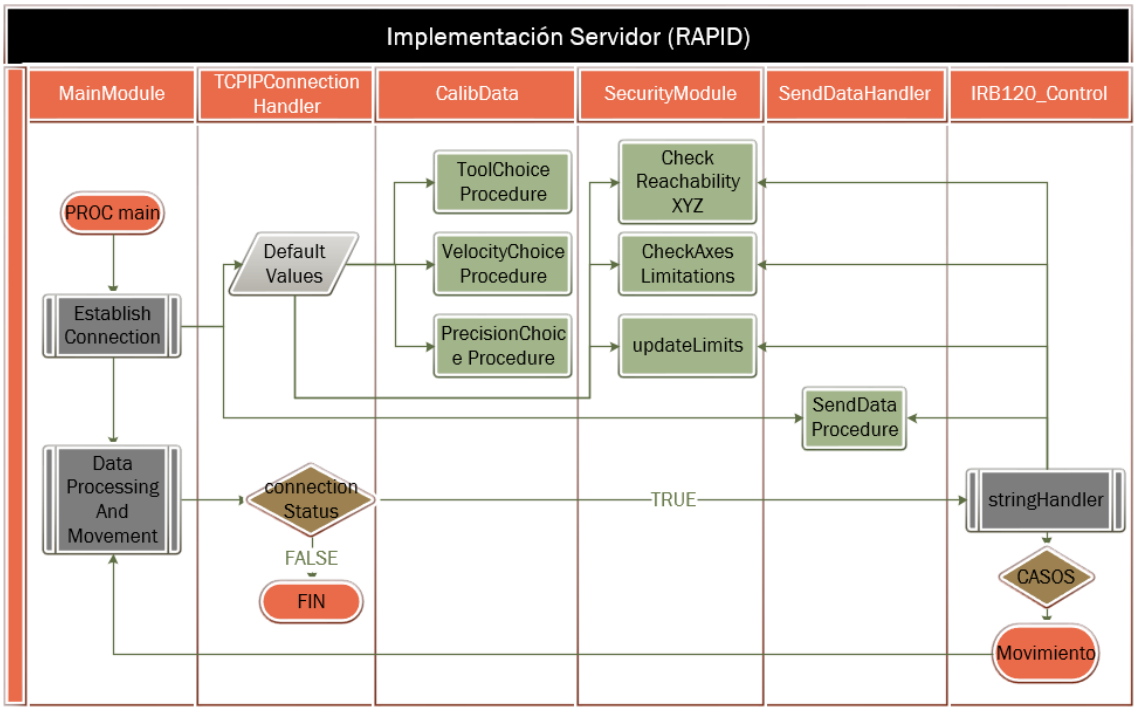


FIGURA 3.2 Esquema de implementación del Servidor

El programa se inicia con el módulo *MainModule* que contiene llamadas a los procedimientos *EstablishConnection* y *DataProcessingAndMovement:*

```
MODULE MainModule

        PROC main()

                EstablishConnection;
                DataProcessingAndMovement;

        ENDPROC

ENDMODULE
```

En el procedimiento *EstablishConnection*, dentro del módulo *TCPIPConnectionHandler* se establece la conexión entre el servidor y el cliente. Para poder realizarse la comunicación, es necesario emplear variables del tipo *socketdev*, en nuestro caso *socketServer* y *socketClient,* que se utilizan para el envío y recepción de datos entre el servidor y el cliente respectivamente, por eso se ha creado como variable global para que todos los módulos puedan tener acceso.

Este procedimiento, en primer lugar, llama a los procedimientos correspondientes dentro de los módulos *CalibData* y *SecurityModule*, para configurar la herramienta seleccionada, la velocidad y la precisión empleadas, y a continuación las limitaciones de los movimientos del robot para poder moverse en su área de trabajo con total seguridad.

A continuación, crea el socket de comunicación (*SocketCreate*) y se enlaza a una dirección IP y a un puerto (*SocketBind*) manteniéndose a la escucha de conexiones entrantes en el puerto especificado (*SocketListen*). El programa se mantendrá a la espera durante un tiempo indefinido (*WAIT_MAX*) hasta que llegue una petición de conexión y sea aceptada (*SocketAccept*). Otra opción sería establecer un tiempo límite para la conexión, y si no se realiza, gestionar los errores que puedan deberse a fallos en la comunicación.

```
MODULE TCPIPConnectionHandler

VAR socketdev socketServer;
VAR socketdev socketClient;
VAR bool      connectionStatus := FALSE;
VAR string    data2ClientType;
LOCAL VAR num       initialVel{2} := [200, 200];

LOCAL VAR string    ipAddress;
LOCAL VAR num       portNumber := 1024;
LOCAL VAR num       err_counter := 0;

  PROC EstablishConnection()

    !============ Default values ==============
    ToolChoiceProcedure 1;
    VelocityChoiceProcedure initialVel;
    PrecisionChoiceProcedure 1;
    pos_limitation.trans.z := 0;  !Define the limitation for the Z axis
    pos_limitation.trans.x := 0;  !Define the limitation for the X axis
    !==========================================

    ipAddress := "127.0.0.1";           !172.22.29.85 for the real station
                                        !127.0.0.1 for simulaiton
    SocketCreate socketServer;
    SocketBind socketServer, ipAddress, portNumber;
    SocketListen socketServer;
    SocketAccept socketServer, socketClient, \ClientAddress:=ipAddress, \Time:=WAIT_MAX;
    connectionStatus := TRUE;

    Target_xyz := CRobT(\Tool:=tool \WObj:=wobj0);

    data2ClientType := "ONLINE";
    SendDataProcedure 0;
    SendDataProcedure 1;


  ENDPROC

ENDMODULE
```

La dirección IP va a depender del modo en el que estemos trabajando, ya que, si estamos en simulación en el propio ordenador la dirección utilizada será '127.0.0.1', mientras que, si estamos trabajando con el robot real, la dirección IP de conexión será '172.29.28.185'.

El procedimiento anterior va a hacer una llamada al procedimiento *SendDataProcedure* dentro del módulo *SendDataHandler* que va a hacer uso de la instrucción *TEST* para evaluar tres casos:

- *CASE 0:* envía al cliente (MATLAB) una cadena de caracteres para asegurar que la conexión se ha establecido. El socket desde el proyecto desarrollado en [3] estaba diseñado para poder establecer una comunicación entre el servidor y el cliente en ambas direcciones, sin embargo, en los otros proyectos mencionados únicamente era el cliente el que mandaba información al servidor. Esta función se ha implementado en el presente proyecto y será objeto de estudio en el siguiente apartado.

```
CASE 0:
        WaitTime 1;
        Data2Client{1} := 0;
        SocketSend socketClient \Data:=Data2Client, \NoOfBytes:=1;

        SocketReceive socketClient \Data:=receivedData \ReadNoOfBytes:=1 \Time:= WAIT_MAX;
        IF receivedData{1} = 0 THEN
          SocketSend socketClient \Str:="Connection has been established successfully";
        ENDIF
```

- *CASE 1*: envía la posición actual de los ejes del robot mediante el array *Data2Client*, sin embargo, en este proyecto no se ha implementado esta función por lo que no se entrará en más detalle.
- *CASE 2*: en función del dato recibido por el cliente, envía un mensaje de error o de conexión satisfactoria.

```
CASE 2:
        Data2Client{1} := 2;
        SocketSend socketClient \Data:=Data2Client, \NoOfBytes:=1;

        SocketReceive socketClient \Data:=receivedData \ReadNoOfBytes:=1 \Time:= WAIT_MAX;
        IF receivedData{1} = 1 THEN

          IF data2ClientType = "ErrPOS_AX" THEN
            SocketSend socketClient \Str:="ERROR: Position of the TCP/axes outside the limit";

          ELSEIF data2ClientType = "ONLINE" THEN
            SocketSend socketClient \Str:="Connection has been established successfully";

          ELSEIF data2ClientType = "ErrROT" THEN
            SocketSend socketClient \Str:="ERROR: Limits exceeded for at least one coupled joint";
          ENDIF

        ENDIF
```

En el módulo *MainModule* observamos que también se llama al procedimiento *DataProcessingAndMovement* que se encuentra dentro del módulo *IRB120_Control,* núcleo principal del programa donde se traduce la información recibida en movimientos del robot.

La función de este procedimiento será leer los datos recibidos en la variable *receivedData* y enviárselo al procedimiento *stringHandler,* dentro del mismo módulo, de manera indefinida, siempre y cuando la conexión sea correcta.

```
PROC DataProcessingAndMovement()
    WHILE connectionStatus DO

        SocketReceive socketClient \Data:=receivedData \ReadNoOfBytes:=1024 \Time:= WAIT_MAX;
         !Receive data from the socket
        stringHandler(receivedData);  !Handle the received data string

    ENDWHILE
   ERROR
    IF ERRNO=ERR_SOCK_CLOSED THEN
        connectionStatus := FALSE;
        SocketClose socketServer;
        WaitTime 2;
        SocketClose socketClient;
        WaitTime 2;
        main;
    ENDIF
  ENDPROC
```

El procedimiento *stringHandler* va a evaluar el primer elemento del dato recibido y en función de éste llevará a cabo una función u otra. Este dato recibido es lo que conocemos como datagrama, y será de vital importancia para establecer una correcta comunicación.

Por tanto, este es un punto clave en el desarrollo del proyecto, ya que como exponía Guillermo Patiño en su proyecto [1], sus aplicaciones de dibujo se encontraban limitadas por el tiempo invertido en la comunicación del socket debido a que en su proyecto cada punto de movimiento del robot era enviado individualmente lo que generaba un retardo en cada trayectoria de unos 3 o 4 segundos, limitación que hacía inviable la reproducción de una imagen con una rapidez elevada.

Por este motivo, se ha diseñado una función en este procedimiento para poder recibir varios puntos a la vez enviados en un mismo datagrama. Para poder entender el desarrollo de la función, primero es necesario detallar la estructura del datagrama que será enviado por MATLAB y que se implementará en el próximo apartado.

El datagrama creado va a poder permitir controlar al mismo tiempo la posición y orientación del TCP (*Tool Central Point*), es decir, va a controlar el movimiento del efector final del brazo robótico, que será un rotulador.

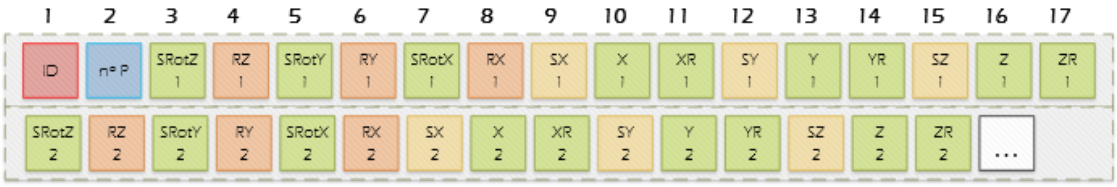


FIGURA 3.3 Estructura datagrama de múltiples posiciones y orientaciones TCP

Como podemos observar los valores correspondientes a la posición y orientación del TCP se envían encadenados uno a continuación de otro, lo que se va a traducir en el robot como una trayectoria de su efector. En la siguiente tabla se detalla el significado de cada uno de los campos:

ID	Nombre del cuadro	Identificador del datagrama
	Tipo de datos	Byte
	Valor	8
	Anotaciones	Contiene el identificador del datagrama que permite utilizar la función creada en el CASE 8 de *stringHandler*.
nº P	Nombre del cuadro	Número de puntos enviados
	Tipo de datos	Byte
	Valor	1-34
	Anotaciones	Determina el número de puntos (posiciones y orientaciones) enviados en cada instante, que será clave para el correcto funcionamiento de la función.
$SROT_{ZYX}\{i\}$/ $S_{XYZ}\ \{i\}$	Nombre del cuadro	Valor de signo
	Tipo de datos	Byte
	Valor	0 o 1
	Anotaciones	Contiene el signo de la posición y orientación del siguiente modo: $\text{signo}\ (J_{n+1}) = \begin{cases} -(J_n + 1) \text{ si } S_n = 0 \\ J_n + 1 \text{ si } S_n = 1 \end{cases}$

R_{ZYX} {i}	Nombre del cuadro	Valor absoluto de la rotación
	Tipo de datos	Byte
	Valor	0-255
	Anotaciones	Contiene el valor absoluto de rotación alrededor del eje EulerZYX correspondiente, respecto al sistema de coordenadas local del TCP. El sistema global tiene la orientación del sistema local (0, 90 ,0).[7]
X/Y/Z {i} / XR/YR/ZR {i}	Nombre del cuadro	Valor absoluto de la posición
	Tipo de datos	Byte
	Valor	0-255 / 0-99
	Anotaciones	Debido a la limitación del tipo byte, es necesario dividir el valor de la posición para alcanzar cualquier posición (mayor de 255 mm) dentro de la zona de trabajo: $\boxed{X_{real} = 100 \cdot X_{frame} + XR}$ Donde: - X_{real} es la coordenada X respecto al sistema robot. - X_{frame} define el número de centenares de X_{real}. - XR define el valor final de 0-99

TABLA 3.1 Descripción campos datagrama de múltiples posiciones y orientaciones TCP

Una vez explicado el tipo de datagrama que va a utilizarse, podemos exponer el *CASE* creado dentro del procedimiento *stringHandler* en el módulo *IRB120_Control*, y es el que se muestra a continuación:

```
CASE 8:
  contador:=0;
  transmission:=0;
  FOR j FROM 1 TO Data2Process{2} DO
  Incr transmission;
    FOR i FROM 3+contador TO 7+contador STEP 2 DO !Valor signo rotacion
      Incr n;
      IF Data2Process{i} = 0 THEN
        rotMatrix{n} := -Data2Process{i+1};
      ELSEIF Data2Process{i} = 1 THEN
        rotMatrix{n} := Data2Process{i+1};
      ENDIF
    ENDFOR
```

[7] Hay que tener en cuenta el orden en el que se realiza la rotación.

```
        n := 0;
        FOR i FROM 9+contador TO 15+contador STEP 3 DO !Valor signo posicion
            Incr n;
            IF Data2Process{i} = 0 THEN
                posMatrix{n} := -(100*Data2Process{i+1} + Data2Process{i+2});
            ELSEIF Data2Process{i} = 1 THEN
                posMatrix{n} := 100*Data2Process{i+1} + Data2Process{i+2};
            ENDIF
        ENDFOR

        effector_orient := OrientZYX(rotMatrix{1}, rotMatrix{2}, rotMatrix{3});
        Target_xyz.rot := effector_orient;

        Target_xyz.trans.x := posMatrix{1};
        Target_xyz.trans.y := posMatrix{2};
        Target_xyz.trans.z := posMatrix{3};

        !====== Security procedure (reachability) ======
        CheckReachabilityXYZ posMatrix;
        !===============================================

        IF statusAVAILABLE THEN
            SingArea \Wrist;
            jtar := CalcJointT (Target_xyz, tool\WObj:=wobj0);
            ConfJ \Off;
            MoveL Target_xyz,vel,fine,tool\WObj:=wobj0;
            SendDataProcedure 1;
        ELSEIF statusAVAILABLE = FALSE THEN
            SendDataProcedure 2;
        ENDIF

        contador:=contador+15;
        n:=0;
        IF Data2Process{2}>1 THEN
            IF transmission=1 THEN
                SocketSend socketClient \Str:="Start";
            ENDIF
        ENDIF
    ENDFOR
```

Todo el *CASE* se va a repetir tantas veces como número de puntos se haya enviado por el datagrama (*Data2Process{2}*), por lo que para la siguiente descripción se tendrá en cuenta el envío de una posición, ya que para el resto es equivalente. Podemos dividir el código en 4 bloques:

- **Cálculo del signo de la rotación del efector:**

Evalúa las posiciones 3, 5 y 7 del datagrama, que se corresponden con los campos *SRotZ, SRotY* y *SRotX* respectivamente. Establece el signo negativo si una de estas posiciones del datagrama es 0, o el positivo si es 1, en las posiciones 4, 6 y 8, correspondientes al valor absoluto de la rotación.

- **Cálculo del signo de la posición del efector:**

Evalúa las posiciones 9, 12 y 15 del datagrama, que se corresponden con los campos *SZ, SY* y *SX* respectivamente. Establece el signo negativo si una de estas posiciones del datagrama es 0, o el positivo si es 1, en las posiciones 10, 11, 13, 14, 16 y 17 correspondientes al valor absoluto de la posición, que como explicábamos se encontraba dividido en dos variables.

- **Envío del movimiento al robot:**

Para enviar la trayectoria que debe seguir el robot se emplea la instrucción *MoveL* que permite el desplazamiento entre diferentes puntos de manera lineal, requisito básico para poder pintar sobre el papel. En un principio se planteó desarrollarlo con la instrucción *MoveJ* que permite un movimiento más rápido de los ejes, sin embargo, esta instrucción al no seguir una trayectoria lineal para alcanzar dos puntos consecutivos de dibujo, el robot se levantaba del papel, o lo que es peor, atravesaba la mesa de trabajo (trayectoria imposible debido a la gestión de errores en el programa) por lo que resultó inviable.

- **Preparación para el envío del siguiente punto:**

La última parte de esta sección de código actualiza el valor de la variable *contador* sumándole 15 unidades para la evaluación del siguiente punto. Esto se debe a que el datagrama enviado para el primer punto consta de 17 posiciones, 2 más que para los siguientes porque incluye el identificador y el número de puntos, que sólo son enviados una vez.

Además, se incluye el envío al cliente a través del socket de la cadena de caracteres *Start* que como veremos más adelante, será empleada por el programa desarrollado en MATLAB para mandar la orden de envío del siguiente conjunto de puntos.

3.3. Implementación del cliente

La parte del socket correspondiente al cliente fue desarrollada por Azahara en su proyecto [2], y en concreto su implementación de la clase de MATLAB *irb120* será la parte que utilicemos como punto de partida en el presente trabajo.

Esta clase emplea funciones que permiten enviar una instrucción de movimiento al robot (bien por RobotStudio o bien al robot real). Sin embargo, la forma en la que se encontraba implementado únicamente permitía enviar una posición cada vez, por este motivo, junto con el desarrollo en el código de RAPID que se explica en el apartado anterior, podemos crear nuevas funciones que se ajustan a nuestras necesidades para poder enviar varias posiciones a la vez.

Para el desarrollo del código del cliente se hará uso de las funciones descritas en la toolbox de control de instrumentos en el apartado 2.3.1.

En primer lugar, se ha modificado la función que permitía la conexión del robot, implementándola del siguiente modo:

```
function connect(r)
   r.conexion=tcpip(r.IP, r.port);
   set(r.conexion, 'Timeout', 2);
   fopen(r.conexion);
   D0=uint8([0 1 0 0 2 1 2 1 0 1 0 1 0]);
   %Enviamos el dato
   fwrite(r.conexion,D0);
   %Leemos conexión
   data = fread(r.conexion);
   string=char(data)'
end
```

Se ha añadido la sentencia de *set* que permite establecer un tiempo de espera máximo de 2 segundos para crear la conexión, tiempo suficiente que reduce la espera, ya que antes no se encontraba limitado.

En el datagrama enviado con el nombre *D0* se ha modificado la primera posición poniendo un 0, que como veíamos en el apartado anterior en RAPID en el procedimiento *SendDataProcedure* dentro del módulo *SendDataHandler,* permitía que el servidor enviara un mensaje al cliente informando que la conexión se había establecido correctamente.

Por último, se emplea la función *fread* para poder leer la sentencia anterior que será enviada en formato columna como una sucesión de números que se corresponden con los caracteres del código ASCII. Por lo que empleando la función *char,* y posteriormente la trasposición (convertir el formato columna en formato fila) del resultado, conseguiremos obtener en la pantalla de MATLAB el citado mensaje.

La siguiente función mostrada es la que corresponde con el envío simultáneo de sucesivas posiciones al robot, y que se denomina *TCPTrajectory:*

```
function TCPTrajectory(r,tcps)

            D8=uint8([8 length(tcps)/6]);
            posiciones=3; %Se desplaza por las posiciones del datagrama
            posicion=1;%Se desplaza por las pos y rot

            for i=1:(length(tcps)/6) %Para cada punto de la trayectoria
                tcp(1:6)=tcps(posicion:posicion+5);
                posicion=((i*6)+1); %Se actualiza

                Rz=uint8(round(abs(tcp(1))));
                signorz = sign (tcp(1));
                if signorz == -1
                    sz=0;
                else
                    sz=1;
                end

                Ry= uint8(round(abs(tcp(2))));
                signory = sign (tcp(2));
                if signory == -1
                    sy=0;
                else
                    sy=1;
                end

                Rx= uint8(round(abs(tcp(3))));
                signorx = sign (tcp(3));
                if signorx == -1
                    sx=0;
                else
                    sx=1;
                end

                xr=rem(abs(tcp(4)),100);
                X=floor(abs(tcp(4))/100);
                signopx = sign (tcp(4));
                if signopx == -1
                    spx=0;
                else
                    spx=1;
                end

                yr=rem(abs(tcp(5)),100);
                Y=floor(abs(tcp(5))/100);
                signopy = sign (tcp(5));
                if signopy == -1
                    spy=0;
                else
                    spy=1;
                end

                zr=rem(abs(tcp(6)),100);
                Z=floor(abs(tcp(6))/100);
                signopz = sign (tcp(6));
                if signopz == -1
                    spz=0;
                else
                    spz=1;
                end
```

```
                d8=[sz Rz sy Ry sx Rx spx X xr spy Y yr spz Z zr];

                D8(posiciones:posiciones+14)=uint8(d8);
                posiciones=(i*15)+3;
            end

            %Enviamos el dato
            fwrite(r.conexion,D8);
            pause(1);
        end
```

Esta función va a abstraer de cada conjunto de 6 elementos del vector de puntos enviados (que puede ser hasta 34), una posición que va a enviar al robot. Los tres primeros dígitos de cada conjunto de 6 corresponden con la rotación (respecto a Z, Y, X, respectivamente), mientras que los 3 siguientes con la posición (en X, Y, y Z, respectivamente).

El código va a evaluar los datos introducidos por el usuario indicando si cada rotación o posición son positivas (colocando un 1) o negativas (colocando un 0) al socket.

Por último, se va a actualizar el campo *D8* para que en cada iteración envíe los 15 datos necesarios para que el código RAPID pueda traducirlo en posiciones del robot.

Como se ha mencionado, el máximo número de puntos enviados cada vez es de 34, por lo que cuando se envíen más de 34 puntos (la mayoría de los casos) necesitamos un método de conexión del cliente con el servidor para que este último indique cuándo puede enviarse la siguiente trayectoria, sin que esto afecte al tiempo de ejecución del robot. Por ello se han llevado a cabo 2 funciones más dentro de esta clase:

```
function communicate(r)
   string='';
   while strcmp(string, 'Start')==0
      data = fread(r.conexion);
      string=char(data)'
   end
end

function socketfree(r)
   data = fread(r.conexion);
   string=char(data)'
end
```

La primera función (*communicate*) se va a basar en la lectura de la cadena *Start* que se enviaba en el código desarrollado en RAPID en el procedimiento *stringHandler* dentro del módulo *IRB120_Control* como se ha visto en el anterior apartado.

Esta función, como veremos más adelante, será utilizada en el código de MATLAB como punto de espera cuando se envía un número de posiciones superior al máximo permitido por el datagrama, para permitir un funcionamiento continuo del robot.

La segunda función (*socketfree*) complementa a la primera y se utiliza cuando ya no se van a enviar más puntos al robot. Cuando el servidor envía un mensaje por el socket al cliente, este mensaje se encuentra disponible hasta que se realiza la operación de lectura, a partir de ese momento, el socket queda liberado para un nuevo envío sin que se sobrescriban los datos. De este modo, esta función permite eliminar del socket el último envío de la cadena *Start* para que no afecte al posterior envío de datos de otra aplicación.

4. APLICACIÓN DE DIBUJO CON PHANTOM OMNI

En este apartado se explicará el núcleo principal del proyecto, que consistirá en la comunicación del brazo robot con el dispositivo háptico Phantom Omni con el objetivo de realizar una aplicación.

4.1. Proceso de calibración

Como veíamos en el apartado de *Herramientas Empleadas,* el Phantom Omni tiene una elevada resolución siendo capaz de detectar incrementos de como mínimo 0.055 mm, lo que lo hace un dispositivo de elevada precisión.

Para poder comprobar la exactitud de la medida obtenida debemos demostrar los resultados obtenidos con la función de lectura de posición incluida en la librería Haptik utilizada para la comunicación, y observar si son fiables.

Como el objetivo de esta aplicación es poder reproducir los movimientos del lápiz del dispositivo en movimiento del brazo robótico para reproducir trazos sobre una hoja, vamos a emplear un sencillo algoritmo que nos permita leer posiciones de la punta del lápiz sobre una plantilla milimetrada.

En primer lugar, debido a que la reproducción del movimiento del Phantom se va a llevar a cabo respecto del sistema de coordenadas del brazo robot, tendremos que pasar del sistema de coordenadas por defecto del Phantom al del IRB120.

Como la función empleada para la lectura de la posición se basa en el sistema de referencia cartesiano de la mano derecha, que es el mismo que emplea el brazo robot, y como sólo vamos a utilizar el valor de la posición y no el de la orientación, únicamente debemos realizar asignaciones entre los ejes para que coincidan.

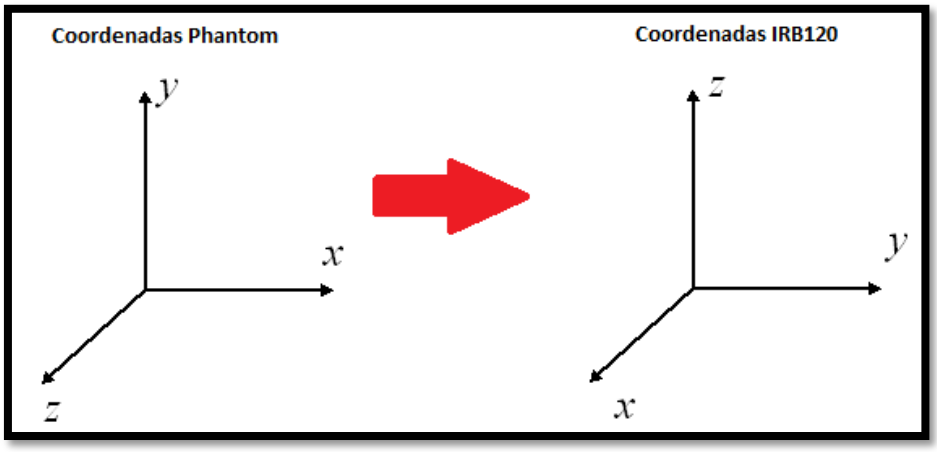


FIGURA 4.1 Cambio de sistema de coordenadas

Así, utilizando el código siguiente comprobamos su exactitud:

```
haptikdevice_list
h=haptikdevice

posiciones=[];
i=0;

while(1)
    i=i+1;
    disp('Selecciona posición y pulsa una tecla...');
    pause;
    pos=read_position(h)
    posiciones=[posiciones; pos(3) pos(1) pos(2)]; %Matriz de vectores fila
    if i==45
        break
    end
end
```

Como habíamos visto, las dos primeras líneas son necesarias para la conexión, después creamos un bucle para poder leer 45 posiciones. Además, la función *read_position* obtiene el valor *x, y, z* respecto al sistema de coordenadas del Phantom, por lo que realizando la lectura en el orden *z, x, y* estaremos en el sistema de coordenadas del robot.

A continuación, se muestra la plantilla que se va a recorrer:

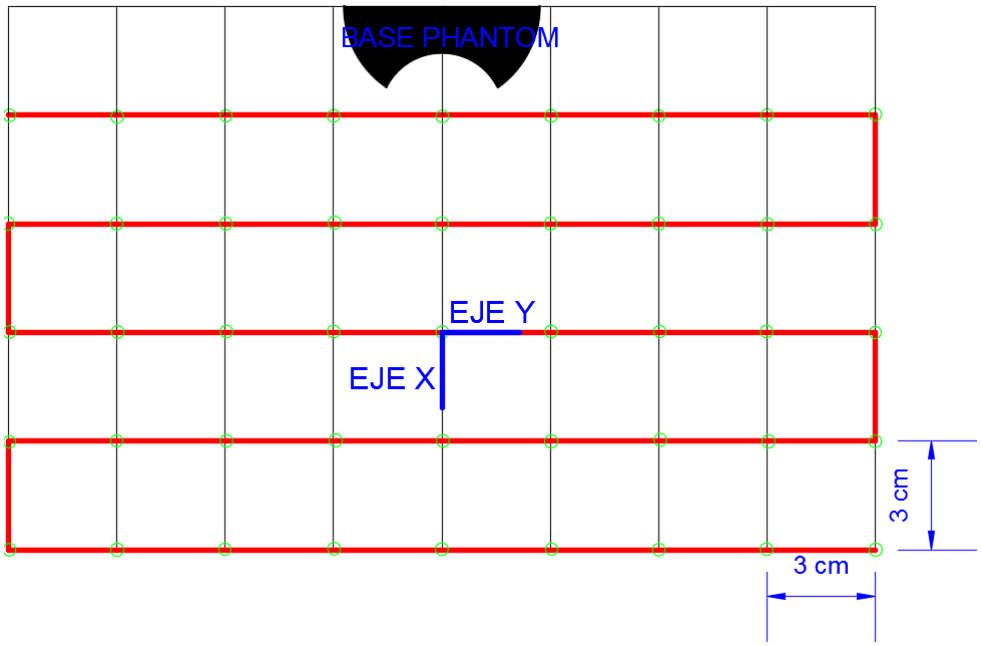


FIGURA 4.2 Plantilla calibración Phantom

Representando los resultados obtenidos en un gráfico tridimensional obtenemos:

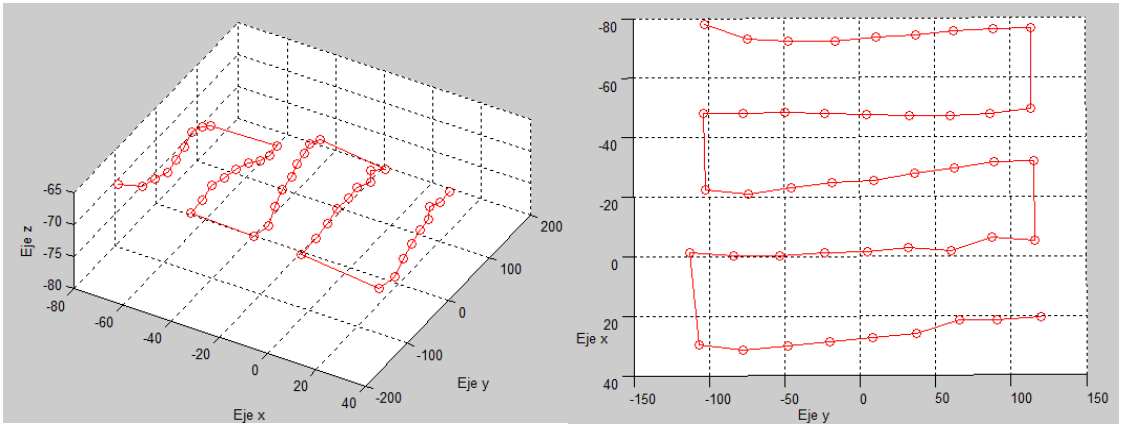


FIGURA 4.3 Calibración posiciones Phantom

Podemos observar que los resultados obtenidos siguen el patrón establecido, pero presentan variaciones, esto se debe a la elevada precisión del dispositivo que hace que una pequeña modificación en la colocación, se traduzca en una diferencia en el resultado. Para comprobar si estos resultados son representativos, realizaremos 2 pruebas más y las compararemos entre sí del siguiente modo.

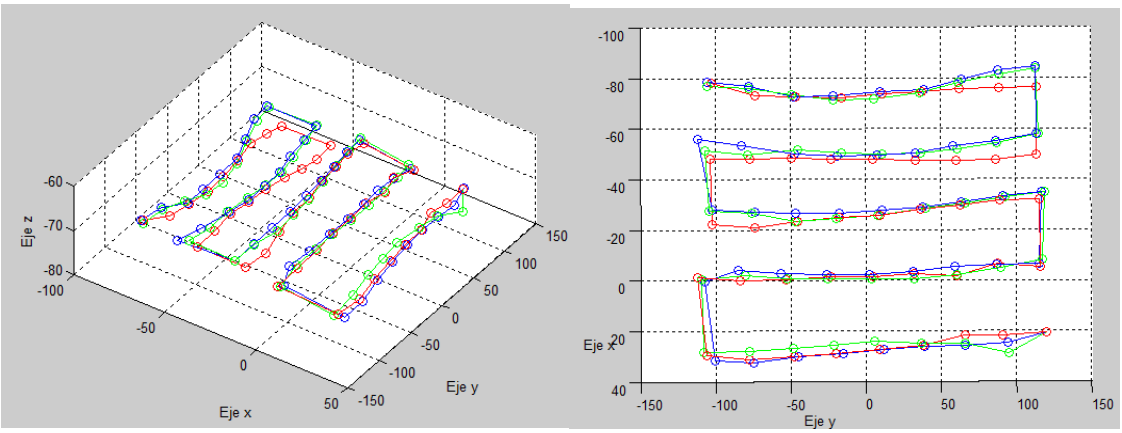


FIGURA 4.4 Calibración posiciones Phantom 2

En el gráfico percibimos que para los mismos puntos obtenemos resultados que no difieren en ningún caso en más de 2 cm, pero que, al no ser exactamente iguales, no nos permite realizar una calibración fija, si no que cada vez que se vayan a adquirir las posiciones deberá calibrarse de nuevo para conseguir una mayor exactitud.

Para realizar el proceso de calibración debemos determinar cuál va a ser el punto de origen de nuestro dibujo. En el proyecto desarrollado en [1] la reproducción tanto de imágenes como textos se realizaba de izquierda a derecha y de arriba abajo, por tanto, el origen *XY* se situaba en la esquina superior izquierda, del mismo modo, se tomará aquí como punto de origen. El origen de la coordenada *z* será establecido donde la punta del lápiz esté en contacto con el papel.

```
%Calibrar inicio dibujo
disp('Origen coordenadas: ')
pause;
pos_ph=read_position(h) %valores en mm

xorigen=pos_ph(3);
x=pos_ph(3)+xcalib

yorigen=pos_ph(1);
ycalib=-yorigen
y=pos_ph(1)+ycalib

zorigen=pos_ph(2);
zcalib=-zorigen;
z=pos_ph(2)+zcalib
```

De este modo con unas simples líneas de código conseguimos que las coordenadas sean positivas de izquierda a derecha en el eje *y,* de arriba abajo en el eje *x* y del plano del papel hacia arriba en el eje *z.*

4.2. Adquisición y envío de la información

El objetivo de esta aplicación es que el usuario pueda dibujar o escribir mediante el Phantom sobre una superficie limitada y que la información de las posiciones por las que ha pasado se almacenen y puedan ser enviadas al robot IRB120 para que éste pueda reproducirlo en el mismo orden en el que el usuario realizó su dibujo.

Para que resulte más intuitiva la explicación, a continuación, se presenta un esquema con los bloques básico para el funcionamiento de la aplicación, donde además se tendrá en cuenta el proceso de calibración visto en el apartado anterior.

FIGURA 4.5 Flujograma de aplicación Phantom

4.2.1. Adquisición del dibujo

Para poder realizar un dibujo mediante el Phantom, se va a aprovechar la posibilidad de obtener su localización espacial además de los botones físicos que incorpora el dispositivo.

Se va a realizar un algoritmo para que el lápiz del dispositivo tenga un funcionamiento idéntico al de un lápiz real, de forma que cada vez que esté en contacto con el papel se encontrará dibujando, mientras que cuando se levante dejará de pintar.

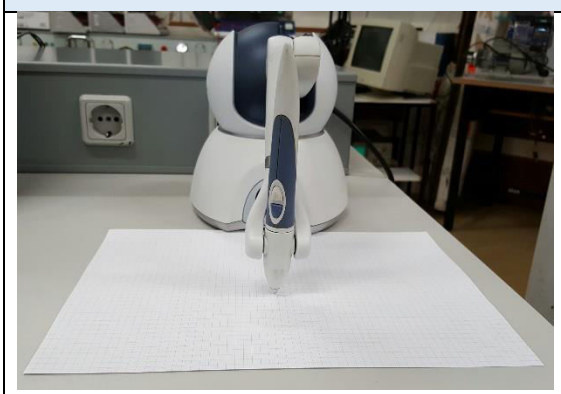
Lápiz sobre papel	Lápiz levantado del papel
	

TABLA 4.1 Posiciones del lápiz del Phantom para dibujar

Idealmente cuando z=0 se considera que se encuentra sobre el papel, mientras que si z>0 estaría por encima y debería dejar de dibujar, sin embargo, como hemos visto, pueden existir pequeñas variaciones que hacen que z pueda no ser 0 y por el contrario encontrarnos sobre el plano de dibujo. Es por ello que se establecerá un margen de seguridad que se deberá tener en cuenta al levantar el lápiz para realizar un nuevo trazo.

Para comenzar a dibujar se pulsará el botón oscuro, mientras que para indicar que el dibujo ha sido finalizado se empleará el botón blanco. Como ya habíamos visto, la función *read_button* nos permite obtener el botón pulsado. El siguiente extracto de código refleja la implementación de la función.

```
while(1)
    while(1)
        pos_ph=read_position(h); %valores en mm
        x=pos_ph(3)+xcalib; x=x*0.75;
        y=pos_ph(1)+ycalib; y=y*0.75;
        z=pos_ph(2)+zcalib;

        button = read_button(h); %1 negro, 2 blanco, 3 ambos

        if z<3 %Margen sobre el papel
           n_puntos=n_puntos+1;

            newline=1;
           pintar=1;

            a=[a_ant,x]; b=[b_ant,y];

            plot(a,b)
            hold on;

            a_ant=x; b_ant=y;

            Matriz_phantom(:,:,n_puntos)=floor([y,x,button,pintar])
            if n_puntos>1 %Para evitar sobreescribir el mismo punto
                if (Matriz_phantom(:,1,n_puntos)==
                    Matriz_phantom(:,1,n_puntos-1))&
                   (Matriz_phantom(:,2,n_puntos)==
                    Matriz_phantom(:,2,n_puntos-1))

                      n_puntos=n_puntos-1; %Para no actualizar la variable
                      break
                end
            end

            if (x>T_img(1) | x<0 | y>T_img(2) | y<0)
                x=0;y=0;
                n_puntos=n_puntos-1;
                break
            end
            break
        end
```

```
        if (z>=3) & (newline==1) %nueva linea  (botón blanco)
            newline=0;
            n_puntos=n_puntos+1;
            pintar=0;

            a=[ ]; b=[ ];
            a_ant=[]; b_ant=[];

            Matriz_phantom(:,:,n_puntos)=floor([y,x,button,pintar])
            break;
        end

        if button==2 %fin del dibujo pulsando blanco
            newline=0;
            n_puntos=n_puntos+1;
            pintar=0;

            a=[ ]; b=[ ];
            a_ant=[]; b_ant=[];

            Matriz_phantom(:,:,n_puntos)=floor([y,x,button, pintar])
            break;
        end
    end

    if button==2 %fin del dibujo pulsando blanco
        break;
    end
end
hold off
```

Para facilitar su comprensión se adjunta el siguiente esquema de funcionamiento del algoritmo:

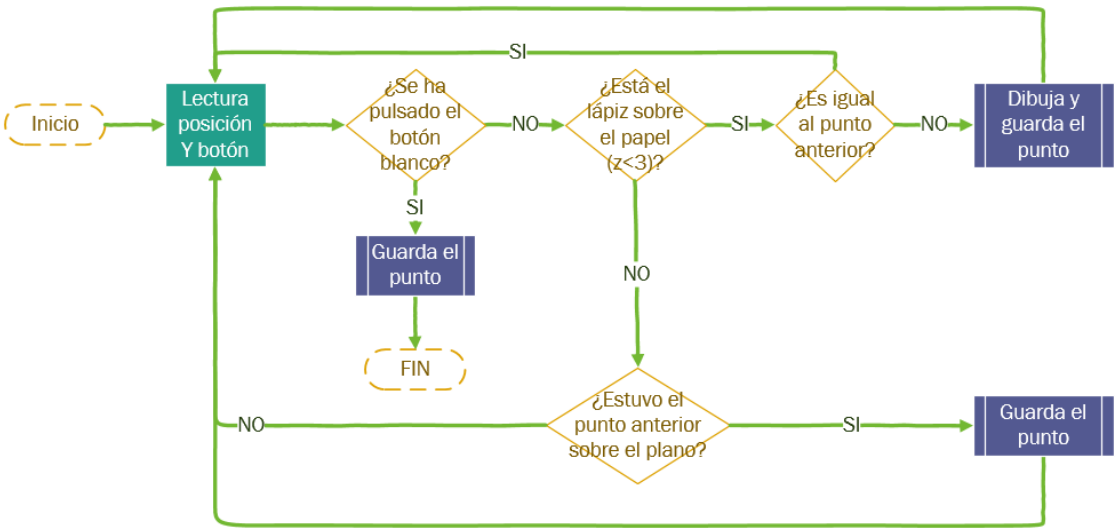


FIGURA 4.6 Esquema de lectura de la posición del Phantom

Se ha creado un bucle que continuamente está leyendo la posición en la que se encuentra el Phantom y en función de la altura y de si se encuentra pulsado algún botón, realizará una acción u otra.

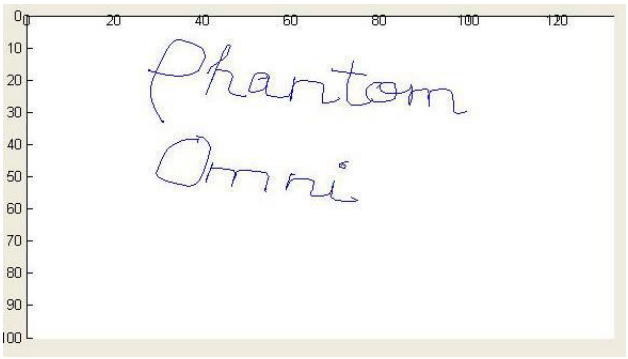


FIGURA 4.7 Imagen obtenida mediante el Phantom

El valor leído por Phantom se obtiene en milímetros, por lo que para poder trabajar, por ejemplo, en una ventana de tamaño 100x100 puntos, y hacer una lectura directamente de la posición del dispositivo, éste sólo permitirá un desplazamiento de 10x10 cm, que deja un espacio de trabajo reducido, es por eso que las variables *x* e *y* se multiplican por un factor de escala de 0.75, para obtener un plano de trabajo un poco más amplio, pero que se encuentre dentro de los límites de trabajo del Phantom.

Como ya se ha mencionado, el Phantom presenta una elevada precisión y nos permite obtener medidas de hasta micrómetros, sin embargo, como para esta aplicación no se requiere tan elevada precisión, mediante la función *floor* redondeamos el valor únicamente a milímetros. Por este motivo, en muchas ocasiones puntos cercanos van a ser leídos como el mismo punto, por ello, se ha creado una condición para que en el caso de que esto ocurra no se vuelva a guardar el punto.

Del mismo modo, cuando se levanta el lápiz no nos interesa que se guarden todos los puntos que recorre el Phantom cuando no se encuentra dibujando, por ello sólo se guardará la posición de reposo producida tras un punto dibujado. Esta condición se lleva a cabo con la actualización de la variable *newline*.

Los puntos de interés serán guardados en una matriz formada por filas con tantas dimensiones como puntos se hayan guardado, respondiendo a la siguiente estructura:

```
Matriz_phantom(:,:,38) =

    74    39     0     1


Matriz_phantom(:,:,39) =

    73    40     0     0


Matriz_phantom(:,:,40) =

    42    41     2     0
```

Donde el primer y segundo elemento corresponden a la coordenada *x* e *y* respectivamente; el tercer elemento indica el valor del botón pulsado, leyendo un 2 si se pulsa el botón blanco (lo que determina el fin del dibujo) o un 0 si no se encuentra pulsado ningún botón (continúa dibujando); y el cuarto elemento corresponde al valor de la variable *pintar* que indicará con un 1 si ese punto debe pintarse o con un 0 si no.

4.2.2. Almacenamiento de los puntos

Una vez guardados los puntos de dibujo en una matriz, se va a realizar un panel de píxeles para identificar las posiciones. Para ello se emplearán 2 matrices homogéneas que representarán los diferentes puntos de dibujo y reposo, con sus mismas coordenadas *XY* respecto al sistema de referencia mundo.

Del mismo modo que se había escogido como punto de origen el vértice superior izquierdo para realizar la calibración, se tomará el mismo punto como punto de referencia para realizar translaciones que correspondan a las coordenadas *XY* de cada punto.

Para que en todo momento el rotulador se sitúe sobre el papel de forma correcta para dibujar, la orientación a la que llega el brazo robot a todos los puntos será perpendicular al papel e indiferente a la del Phantom, ya que, a la hora de dibujar, el lápiz se colocará con la inclinación necesaria que nos permite su control.

Para simplificar, establecemos únicamente 2 posiciones para cada punto en el panel de dibujo: una a la altura del papel, y otra, 5 cm encima; y se crean tantas posiciones como puntos se hayan guardado en la matriz de puntos, como podemos ver en la siguiente imagen:

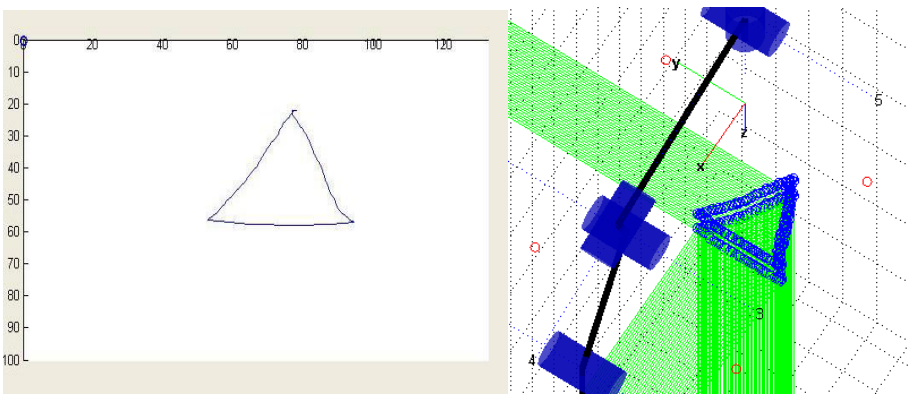


FIGURA 4.8 Ejemplo panel de píxeles creado para aplicación Phantom

En la imagen anterior, puede observarse que aparece encuadrada entre 4 círculos rojos. Estas marcas son utilizadas para establecer los límites de la hoja real de dibujo, que podrá ser introducido por el usuario mediante la interfaz. El código correspondiente a la creación se estos puntos es el siguiente:

```
for i=1:1:4 %Vértices zona dibujo
    if i==2 %Arriba derecha
        L3=Papel*T_img(1);
        L4=0;
    elseif i==3 %Abajo derecha
        L3=Papel*T_img(1);
        L4=Papel*T_img(2);
    elseif i==4 %Abajo izquierda
        L3=0;
        L4=Papel*T_img(2);
    end

    angle=90;
    ROTACION=troty(angle,'deg');
    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);
    hold on

    x1=0;y1=0;z1=0;
    P1=[x1 y1 z1 1]'; T1(:,i)=(Tbe)*P1;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    hold on;
    plot3(u1,v1,w1,'or'); %Dibujo vértices
end
```

La variable *Papel* va a determinar la separación entre píxeles de la imagen, que será objeto de estudio en el desarrollo de la interfaz, mientras que *T_img* contendrá los píxeles de ancho y alto de la imagen respectivamente.

Para la representación espacial del robot es necesario utilizar matrices homogéneas que permitan desplazar el efector final del brazo robot a la posición deseada, de este modo se crea la matriz *Tbe,* que incluirá el desplazamiento respecto a los ejes XY, y después será multiplicada por las coordenadas del punto para determinar su altura.

Para la creación de los puntos que forman parte del dibujo se sigue el mismo procedimiento, sólo que van a crearse dos posibles posiciones como veíamos en el dibujo, diferenciadas únicamente por la altura y que serán guardadas en dos matrices: *Pintar* y *No_pintar*, que son utilizadas respectivamente para el envío de la información y contendrán tantas matrices homogéneas como puntos se hayan guardado en *Matriz_phantom*. El siguiente extracto de código muestra la creación de estas matrices:

```
n_puntos=size(Matriz_phantom);

for i=1:1:n_puntos(3) %Puntos de dibujo
    Puntos=Matriz_phantom(:,:,i)
    L3=Puntos(1)*Papel; %Coordenada x (30cm máximo)
    L4=Puntos(2)*Papel; %Coordenada y (m)

    angle=90; ROTACION=troty(angle,'deg');
    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);

    %Definimos los puntos de nuestro minipanel%
    x1=0;x2=0;
    y1=0;y2=0;
    z1=-0.005;z2=-0.05;

    P1=[x1 y1 z1 1]'; T1(:,i)=(Tbe)*P1;
    P2=[x2 y2 z2 1]'; T2(:,i)=(Tbe)*P2;


    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    Tp1(:,:,i)=[-1 0 0 u1;0 1 0 v1;0 0 -1 w1;0 0 0 1];
    u2=T2(1,i);v2=T2(2,i);w2=T2(3,i);
    Tp2(:,:,i)=[-1 0 0 u2;0 1 0 v2;0 0 -1 w2;0 0 0 1];

    Pintar(:,:,i)=[Tp1(:,:,i)];
    No_pintar(:,:,i)=[Tp2(:,:,i)];
end
```

4.2.3. Envío de la información

Una vez que hemos guardado los puntos en las matrices *Pintar* y *No_pintar,* en este apartado, en función de los valores de la variable *pintar* de *Matriz_phantom* se asignará el envío de un punto al robot correspondiente a la primera o segunda matriz.

En el siguiente fragmento de código podemos ver esa asignación en la variable *numero*:

```
    if j==1 %Primera iteración
        posicion=Matriz_phantom(:,:,j);
        pos_img=j;

        if posicion(4)==1 %Pinta
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);
        elseif posicion(4)==0 %No pinta
            numero=No_pintar(:,:,pos_img);
        end

    else
        posicion=Matriz_phantom(:,:,j); %A partir de la segunda iteración
        posicion_anterior=Matriz_phantom(:,:,j-1);
        pos_img_ant=j-1;
        pos_img=j;
```

```
        if posicion(4)==1 & posicion_anterior(4)==1 %Dos puntos seguidos
            numero=Pintar(:,:,pos_img); %Dibuja Línea
        elseif posicion(4)==0
            numero=No_pintar(:,:,pos_img_ant);
        elseif posicion(4)==1 & posicion_anterior(4)==0 %Empezar a pintar
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);
        elseif posicion(3)==2
            numero=No_pintar(:,:,pos_img);
        end
    end
```

Se realizarán tantas iteraciones como número de posiciones tenga guardada la *Matriz_phantom*. El procedimiento de evaluación será el mismo para cada punto salvo para el primero que únicamente se valorará si debe pintar o no y en consecuencia actualizar la variable *numero*.

Para el resto se guardarán la posición actual que se está evaluando en *posición* y la del punto anterior en *posición_anterior*, y se actualizará la variable *numero* en función de 4 condiciones:

1. **El punto actual debe pintarse y el anterior se ha pintado**. En este caso el rotulador debe mantenerse sobre la hoja sin levantarse, por lo que únicamente se enviará la siguiente posición a pintar.
2. **El punto actual no debe pintarse**. En este caso únicamente se guardará la matriz correspondiente a *No_pintar,* pero respecto a la última posición y no a la actual, ya que la acción que se enviará será la de levantar el rotulador verticalmente sobre el último punto dibujado. En esta sentencia no es necesario incluir la condición de que el último punto fuera sobre el papel, ya que como veíamos en el apartado de *Realización del dibujo* sólo va a guardarse la posición de reposo tras un punto dibujado, pero en caso contrario habría que haberlo incluido.
3. **El punto actual debe pintarse, pero el anterior fue un punto de reposo**. A diferencia del caso anterior, en este punto es necesario establecer ambas condiciones. Como el punto anterior se encontraba encima del papel sobre el último punto dibujado, la nueva acción será situarse encima del siguiente punto que debe dibujarse y, a continuación, bajar hasta el papel y pintarlo.
4. **Se ha pulsado el botón blanco**. Como ya se ha explicado en apartados anteriores, la función del botón blanco será la de determinar el fin del dibujo, y por tanto cuando se pulse, se enviará una posición de reposo sobre el último punto dibujado.

A partir de aquí trabajaremos con la posición o posiciones que se han guardado en la variable *numero*. En las siguientes líneas de código podemos observar el formato de envío de la información:

```
    for int=1:1:((length(numero)/4))
         Tpos=numero(1:4,1+a:4+a)
         pos=transl(Tpos);
         pos=pos*1000  %posiciones a milímetros
         rot=tr2rpy(Tpos);
         rot=rot*180/pi %ángulos a grados
         tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)]

         posi=posi+1;
         a=a+4;
    end
```

Se crea un bucle *for* que tendrá tantas iteraciones como matrices homogéneas contenga la variable *numero*, que será 1 o 2. En el caso de tener 2 matrices, como se guardan en las mismas filas, pero columnas consecutivas, será necesario evaluarlas por separado.

Obtendremos la posición y orientación de cada punto y lo guardaremos en filas en la matriz *tcp* en el orden establecido por la función creada para la comunicación mediante el socket entre MATLAB y el robot.

Como ya se mencionó, dicha función es capaz de enviar simultáneamente 34 puntos de modo que el robot siga en línea recta la trayectoria correspondiente. Para aprovechar su máxima capacidad, cada vez que se guarde un punto en *tcp*, se actualizará la variable *posi*, y cuando ésta llegue al valor de 34, serán enviadas las posiciones al robot.

```
if posi>=34 %34puntos máximo
        cont=1;
        for i=1:(posi-1) %Número de puntos
            tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
            cont=(i*6)+1;
        end

        robot.TCPTrajectory(tcps)%Dibuja las líneas

        %Se mantiene a la espera hasta enviar siguiente trayectoria
        if orden<=n_lineas(3)
            robot.communicate
        else
            robot.socketfree;
        end

        posi=1;

        %Borrar en cada iteración
        tcps=[]; numero=[];
end
```

Sin embargo, el vector *tcp* no puede enviarse tal cual se encuentra estructurado, ya que como recordamos el envío de posiciones simultáneas se realiza de forma concatenada, por ese motivo, mediante un nuevo bucle *for* reordenamos las posiciones y orientaciones de los 34 puntos en la variable *tcps*.

Tras el envío de las 34 posiciones se mantendrá a la espera para la 34 siguientes mediante *robot.communicate*, o en caso de no haber más posiciones de envío, se liberará el socket mediante *robot.socketfree* para una nueva adquisición. Es importante que tras cada envío se borre la variable *numero* para evitar sobrescribir datos anteriores ya que sus dimensiones pueden variar.

Como es improbable que un dibujo necesite exactamente 34 puntos o un múltiplo de 34, se ha creado adicionalmente la posibilidad de enviar menos posiciones en el caso de resten menos de 34 puntos para finalizar el dibujo. El modo de operación es idéntico al que se acaba de explicar:

```
%Cuando quedan menos de 34 puntos
if posi>1
    tcps=[];
    cont=1;
    for i=1:(posi-1) %Número de puntos de Tpos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca las q en una fila
        cont=(i*6)+1;
    end
    robot.TCPTrajectory(tcps)%Dibuja los puntos
    robot.socketfree;
end
```

4.3. Resultados

En este apartado se mostrarán los resultados obtenidos en el brazo robot IRB120 y en el simulador. Se debe prestar atención a que en los resultados obtenidos en simulación aparecen las líneas correspondientes a los trazos de reposo, que en el dibujo sobre el papel no quedarían marcados al realizarse por el aire.

Con esta aplicación podemos tanto escribir textos como dibujar imágenes a mano alzada, o reproducirlas siguiendo una plantilla, pero en cualquiera de los casos debe realizarse un correcto proceso de calibración para conseguir los resultados más precisos.

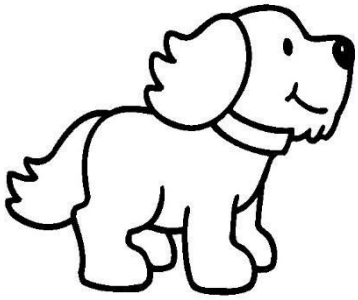
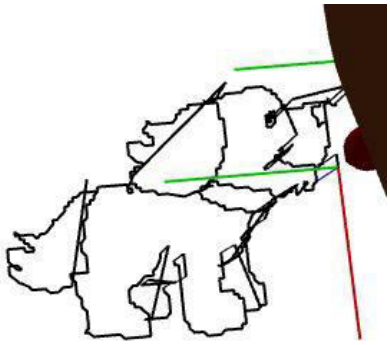
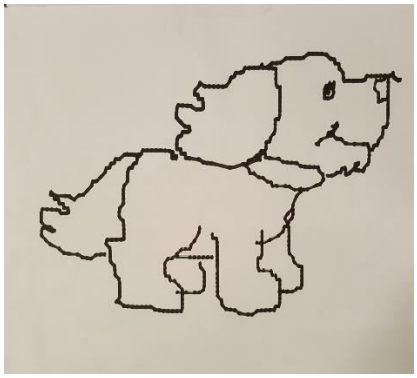
Modelo plantilla	Copia a mano alzada con Phantom
	
Simulación RobotStudio	Resultado obtenido con IRB120
	

TABLA 4.2 Resultado copia de modelo de imagen perro

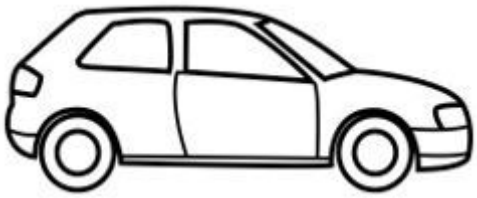
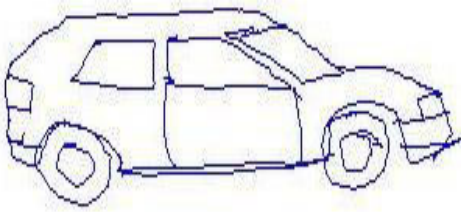
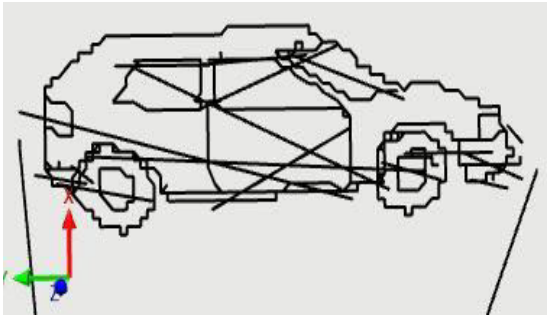
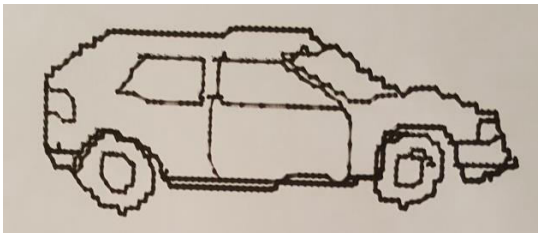
Modelo plantilla	Copia a mano alzada con Phantom
	
Simulación RobotStudio	Resultado obtenido con IRB120
	

TABLA 4.3 Resultado copia de modelo de imagen coche

5. MEJORA EN REPRODUCCIÓN DE IMÁGENES Y TEXTOS

En este apartado se abordarán las modificaciones realizadas sobre el proyecto desarrollado por Guillermo en [1].

5.1. Punto de Partida

El proyecto "Reproducción de imágenes y textos con el robot IRB120" [1] tiene como objetivo el desarrollo de varias aplicaciones de escritura y dibujo en MATLAB, para que, a partir del socket desarrollado en [2], pueda comunicarse con el robot IRB120, y éste, mediante un rotulador colocado en el efector final, pueda pintar sobre una hoja la información que se le envía en forma de dibujo, letras y/o números.

En concreto, este proyecto consta de 4 aplicaciones que explicaremos en los siguientes puntos. Cabe destacar que debido a que esta parte del trabajo se basa en una mejora de un trabajo ya existente, únicamente se desarrollará en profundidad los cambios introducidos, y se expondrá de forma concisa la principal funcionalidad del proyecto que tomamos como base para facilitar al lector la comprensión de las modificaciones.

Al presentar todas las aplicaciones desarrolladas la misma finalidad (envío de información que se traduzca en dibujo sobre un papel) pero con diferentes funcionalidades, van a tener una estructura común, por ello resulta más sencillo representarlo mediante el siguiente esquema:

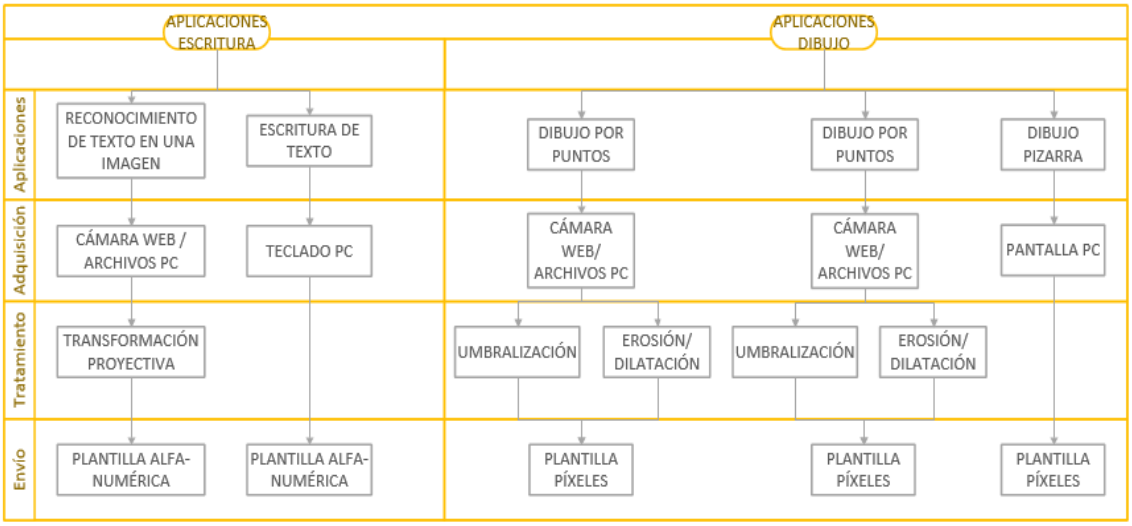


FIGURA 5.1 Esquema de aplicaciones

5.2. Aplicaciones

5.2.1. Aplicaciones de escritura

Dentro de este apartado podemos encontrar dos aplicaciones, pero debido a la similitud serán explicadas a la vez, ya que la diferencia se encuentra en la forma de obtener el texto que se va a enviar.

- **Adquisición:**

Para poder reconocer el texto de una imagen, primero es necesario que pueda ser adquirida por el usuario. En el proyecto anterior se plantearon 2 formas de realizarlo, que se han mantenido en el presente trabajo:

- Mediante una cámara web: haciendo uso de funciones proporcionadas por MATLAB se puede establecer la conexión con una cámara integrada, o externa y hacer una captura:

```
webcamlist; %Busca las cámaras conectadas
cam=webcam(1); %Se establece una cámara
preview(cam); %Previsualización imagen
closePreview(cam); %Cerramos la previsualización
img=snapshot(cam); %Toma una imagen
imshow(img); %Muestra la imagen por pantalla
```

- Archivos del propio equipo: del mismo modo que antes, MATLAB contiene funciones que permiten realizar una búsqueda de imágenes en el propio equipo:

```
img=imgetfile; %Examina el equipo
imshow(img); %Muestra la imagen por pantalla
```

- **Tratamiento:**

Este punto es importante a la hora de adquirir el texto de una imagen para poder representarlo, ya que en muchas ocasiones el texto no se presentará totalmente recto, y esto dificultará la correcta lectura de la imagen por parte de las funciones de MATLAB.

Las transformaciones geométricas modifican la relación espacial entre píxeles. En imágenes 2D podemos emplear transformaciones proyectivas que permiten cambiar la forma de las imágenes y proyectarlas desde diferentes ángulos para obtener el resultado deseado. Gracias a la toolbox de procesamiento de imágenes de MATLAB encontramos funciones que permiten realizar fácilmente este tipo de transformación:

```
[x, y]=ginput; %permite recuadrar la zona de interés de la imagen
tform=maketform ('projective', [x(1) y(1); x(2) y(2); x(3) y(3); x(4)
y(4)], ...
                                  [0 0; 1.5 0; 0 1; 1.5 1]);
[img, x, y]=imtransform(imagen, tform, 'bicubic', ...
                        'udata', udata, ...
                        'vdata', vdata, ...
                        'size', size(imagen), ...
                        'fill', 128);
```

- **Almacenamiento y envío:**

Una vez que la imagen ha sido tratada correctamente, se pueden utilizar las funciones de la toolbox de Visión por Computadora de MATLAB para poder reconocer el texto que se encuentra en ella.

```
imshow(img); %Se muestra la imagen
roi=round(getPosition(imrect)); % Recuadra la región de interés
ocrResult=ocr(img, roi);
Iocr=insertText(img, roi(1:2), ocrResult.Text, 'AnchorPoint', ...
                'RightTop', 'FontSize', 16); %Muestra texto
```

Debido a que los caracteres que van a ser representados se corresponden con las letras del alfabeto y los dígitos del 0 al 9, la solución más intuitiva es crear una plantilla alfa-numérica definiendo previamente las trayectorias que debe seguir el efector final para cada carácter a partir de unos puntos establecidos sobre la plantilla.

En la imagen podemos observar que aparecen dos alturas, una se corresponde con la posición de dibujo, y otra con la de reposo para levantar el rotulador.

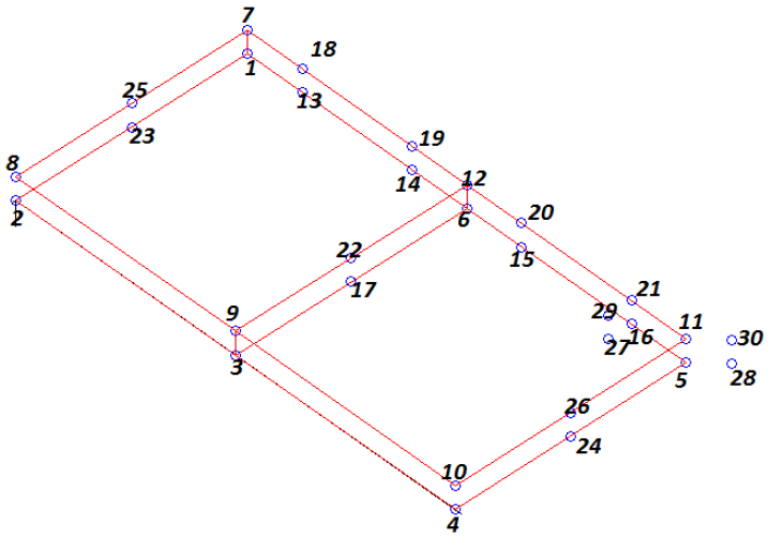


FIGURA 5.2 Panel individual de representación de caracteres

La solución llevada a cabo en el anterior trabajo fue la de crear tantos paneles individuales de cada carácter como número de letras y/o números tuviera el texto que se fuera a escribir, y a la hora de enviar la información al robot, únicamente acceder a la posición correspondiente de cada carácter. Así cada carácter estará formado por n-vectores de puntos n-dimensionales, siendo n el número de posiciones ocupadas por el texto.

```
for i=1:1:(n_filas*n_letras_fila)
                . . .

    letraa(:,:,i)=[Tp10(:,:,i) Tp4(:,:,i) Tp2(:,:,i) Tp1(:,:,i) Tp5(:,:,i)
     Tp11(:,:,i) Tp10(:,:,i) Tp9(:,:,i) Tp3(:,:,i) Tp6(:,:,i) Tp12(:,:,i)];
                . . .
end
```

Aunque a priori parece que existe un trabajo innecesario debido a que se van a crear paneles que no van a ser utilizados, esto no afecta al tiempo de ejecución, por lo que en el presente proyecto se ha optado por continuar con esta solución.

Para poder leer cada uno de los caracteres reconocidos en una imagen o escritos mediante teclado, se empleará la función *abs* que permite obtener el código ASCII correspondiente a cada carácter de una cadena. De este modo, se evalúa el código de cada letra y/o número y se le asigna la trayectoria creada en el panel y en la posición correspondiente.

A partir de este punto es donde se introducen los cambios más significativos. En primer lugar, en el anterior proyecto quedaba definida una trayectoria para el carácter *espacio* que consistía en situar el efector encima del panel sin pintar hasta que se mandara la siguiente trayectoria, lo que suponía una pérdida de tiempo ya que no se representaba nada en el papel. Para hacerlo más eficiente, se ha eliminado esta trayectoria de forma que cuando haya espacios de por medio el efector saltará tantas posiciones como espacios haya e irá directamente a la trayectoria del carácter inmediatamente después de ellos.

```
for j=1:1:length(cadena) %Para cada caracter
    a=0;
    if abs(cadena(j))==97 |abs(cadena(j))==65
        numero=letraa(:,:,orden);
        orden=orden+1;
                . . .

    elseif abs(cadena(j))==32  %Espacio
        orden=orden+1;
        space=1;
                . . .
    end

    posi=1;
    tcp=[];

    for int=1:1:((length(numero)/4))
        Tpos=numero(1:4,1+a:4+a); %recorre cada vez 4 columnas
        pos=transl(Tpos); pos=pos*1000;  %posiciones a milímetros
        rot=tr2rpy(Tpos); rot=rot*180/pi; %ángulos a grados
        tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)];

        posi=posi+1; a=a+4;
    end

    cont=1;
    for i=1:(posi-1) %Número de puntos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
        cont=(i*6)+1;
    end

    %Se mantiene a la espera hasta enviar siguiente trayectoria
    if space==0
        robot.TCPTrajectory(tcps);
        robot.communicate;
    else
        space=0;
        robot.socketfree;
    end

    numero=[];tcp=[];tcps=[];
end
robot.socketfree;
```

La variable *numero* va a contener tantas matrices homogéneas (4x4) como puntos tenga el carácter que se va a representar, que serán abstraídos por la variable *Tpos* guardando su posición y orientación en filas de vectores, dentro la matriz *tcp*. Debe recordarse que la orientación está definida para ser siempre perpendicular al área de dibujo.

Del mismo modo que veíamos en la aplicación de Phantom, las filas de la matriz *tcp* serán pasadas al vector *tcps* donde se almacenarán consecutivamente para poder ser enviadas por la función *robot.TCPTrajectory* como habíamos descrito en apartados anteriores. De este modo, somos capaces de enviar todas las posiciones de cada carácter a la vez.

- **Resultados:**

En este punto se mostrarán algunos resultados obtenidos. Debe tenerse en cuenta que el resultado plasmado en el papel es idéntico al del proyecto que toma como base, pero, por cada punto de una letra, en este caso se realizan 4, reduciéndose el tiempo de ejecución total en una cuarta parte. Además, los resultados obtenidos en simulación muestran tanto las trayectorias de dibujo como las de reposo, pero el resultado final únicamente reflejaría el trazo del rotulador sobre el papel.

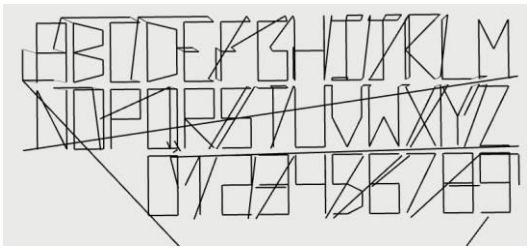
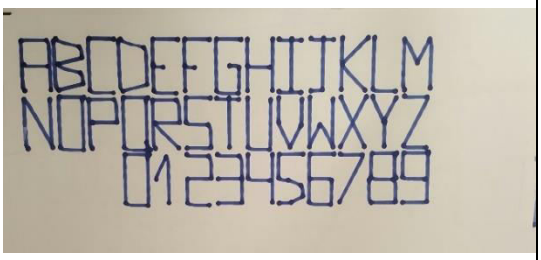
Simulación en RobotStudio	Resultado obtenido IRB120
	

TABLA 5.1 Resultados reproducción texto vía teclado

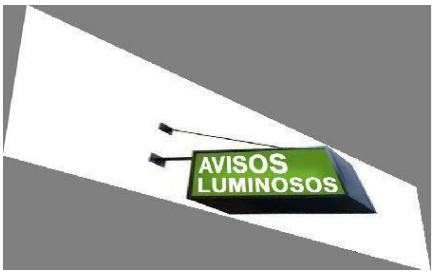
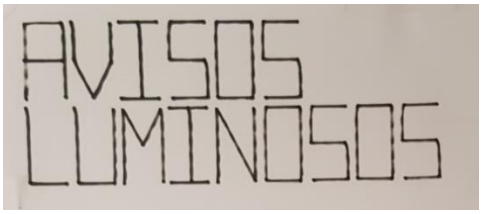
Imagen adquirida	Imagen transformada
	
Simulación RobotStudio	**Resultado obtenido con IRB120**
	

TABLA 5.2 Resultados reproducción texto mediante reconocimiento de imagen

5.2.2. Aplicaciones dibujo

Dentro de este apartado podemos encontrar 3 aplicaciones, dos de las cuales (dibujo por puntos y dibujo por líneas) presentan puntos en común que expondremos a continuación.

El proceso de adquisición tanto para el dibujo por líneas como el de por puntos es el mismo que en el caso anterior de reconocimiento de texto en una imagen por lo que no se volverá a explicar.

- **Tratamiento**

Este punto es común para las dos aplicaciones mencionadas debido a que el dibujo mediante pizarra no requiere ningún tratamiento. Para este apartado serán necesarias las funciones que incluye la toolbox de Procesamiento de Imágenes de MATLAB:

- Segmentación de imágenes: es un proceso que se encarga de dividir una imagen digital en grupos de píxeles con el objetivo de poder simplificar o modificarla para que resulte más fácil de analizar. Existen diferentes técnicas de segmentación, pero en el trabajo anterior se empleó la umbralización.

  - Umbralización: consiste en segmentar la imagen agrupando los píxeles que tienen características similares, *binarizando* la imagen en dos segmentos: un fondo (color blanco) y un objeto (color negro). La asignación de un pixel a uno de los segmentos se consigue comparando su nivel de gris con un valor umbral preestablecido (*threshold*). Esta técnica resulta de vital importancia en el desarrollo de estas aplicaciones, a pesar de que esto conlleva una significativa reducción de los detalles de la imagen, debido a que el efector final del brazo robot únicamente va a disponer de un color para dibujar.

```
imgray=rgb2gray(imagen)%Convierte imagen de color a escala de grises
imgbw=im2bw(imagen, 0.5) %Convierte la escala de grises a 2 valores
```

- Operaciones morfológicas: sirven para simplificar imágenes binarias (*umbralizadas*) conservando las principales características de forma de los objetos con el objetivo de reducir el ruido de la imagen o definir la estructura de los objetos agrupando o aislando elementos. Las dos operaciones morfológicas fundamentales son la dilatación y la erosión:

- Dilatación: es una transformación que consiste en que si alguno de los píxeles que están próximos al pixel de estudio pertenecen al segmento objeto, el pixel de estudio también pertenece al segmento objeto. Esta operación permite añadir píxeles al segmento objeto que hayan podido perderse en la umbralización.
- Erosión: es una transformación que consiste en que el pixel de estudio pertenecerá al segmento objeto únicamente si todos los píxeles próximos pertenecen al segmento objeto, en caso contrario, pertenecerá al fondo. Mediante esta operación se elimina información innecesaria del segmento objeto.

```
se1=strel('line', 2, 0); %Crea las filas de la plantilla de píxeles
se2=strel('line', 3, 90); %Crea las columnas de la plantilla de píxeles

dilatacion=imgdilate(imagen, [se1,se2], 'full'); %Dilata la imagen
erosion=imerode(imagen, [se1,se2], 'full'); %Erosiona la imagen
```

5.2.2.1. Aplicación dibujo por puntos

Una vez que la imagen ha recibido un tratamiento y se encuentra en formato binario, mediante la aplicación por puntos se va a adquirir el conjunto de píxeles que forman parte del segmento objeto para poder reproducir la imagen.

- **Almacenamiento**

En el trabajo anterior se propuso una sencilla pero eficiente forma de obtener los píxeles pertenecientes al objeto, necesarios para realizar el dibujo, descartando los del fondo. En el siguiente esquema se explica la metodología:

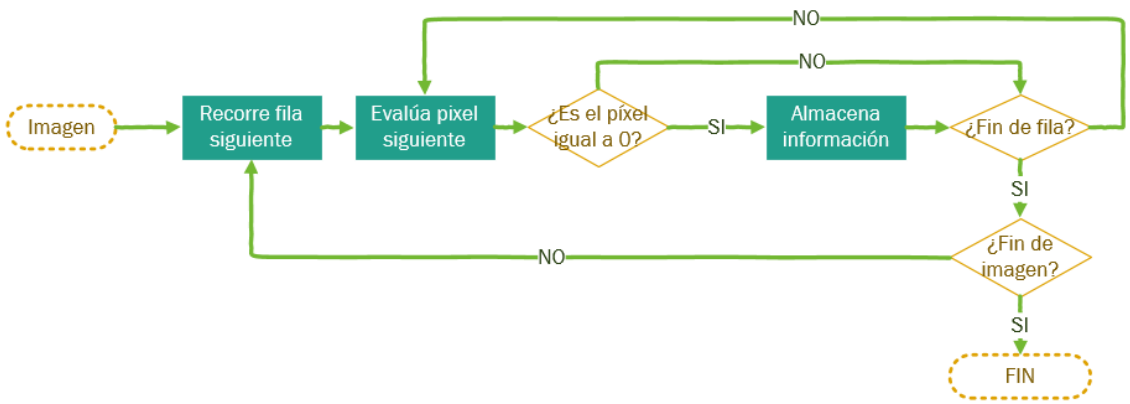


FIGURA 5.3 Almacenamiento de la información por puntos

Una vez almacenados los píxeles del segmento objeto, tienen que convertirse en puntos de dibujo que reproduzca el robot. En el proyecto desarrollado en [1] se proponía como solución crear una plantilla de píxeles con tantos puntos como píxeles totales tuviera la imagen, sin embargo, en este proyecto, se ha considerado innecesario y poco eficiente crear puntos que no van a ser dibujados por el robot, y por ello se ha optado por guardar únicamente las posiciones en las que va a dibujar el robot, con lo que se consigue ahorrar tiempo en el preprocesado ya que evalúa únicamente la información de interés.

```
Puntos=Matriz_puntos(:,:,i);
L3=Puntos(2)*Papel; %Ancho
L4=Puntos(1)*Papel; %Alto
Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*transl(0,-L3,0)*transl(L4,0,0);
```

Se recorre la matriz donde se encuentran guardados los pixeles de dibujo, la cual contiene en el primer elemento la coordenada vertical y en el segundo, la horizontal.

Las variables *L3* y *L4* corresponden al desplazamiento en horizontal y vertical respectivamente, considerando como origen de coordenadas la esquina superior izquierda de la imagen; mientras que la variable *Papel* realiza una conversión entre las coordenadas de la imagen y las del papel de dibujo estableciendo unas dimensiones para cada punto en el papel previamente elegidas por usuario, como se verá en el desarrollo de la interfaz.

- **Envío**

Del modo en el que se ha establecido el almacenamiento de la información, tendremos una matriz formada por pares de puntos (punto de dibujo y punto de reposo) correspondiente únicamente a los píxeles del segmento objeto, que serán los que se evalúen y se conviertan en posiciones que se enviarán al robot.

```
n_puntos=size(Matriz_puntos);
pos=1;orden=1;
              . . .

    pos_img=j;
    numero=Punto(:,:,pos_img);

    for int=1:1:((length(numero)/4))
        Tpos(:,:,pos)=numero(1:4,1+a:4+a);
                      . . .
        tcp(pos,:)=[rotation(3) rotation(2) rotation(1) position(1)
                    position(2) position(3)]
        a=a+4;
        pos=pos+1;
    end
```

El envío de la información sigue la misma estructura que para la aplicación Phantom, siempre que queden suficientes puntos para enviar, se aprovechará el tamaño máximo que nos permite *TCPTrajectory* (34 posiciones) lo que permitirá una mejora sustancial en la velocidad de dibujo respecto al proyecto [1].

- **Resultados**

En este punto se mostrarán algunos resultados obtenidos. Debe tenerse en cuenta que el resultado plasmado en el papel es idéntico al del proyecto que toma como base, pero, por cada punto de la imagen, en este caso, se dibujan 3 puntos, reduciéndose el tiempo de ejecución total en un tercio. Además, los resultados obtenidos en simulación muestran tanto las trayectorias de dibujo como las de reposo, pero el resultado final únicamente reflejaría el trazo del rotulador sobre el papel.

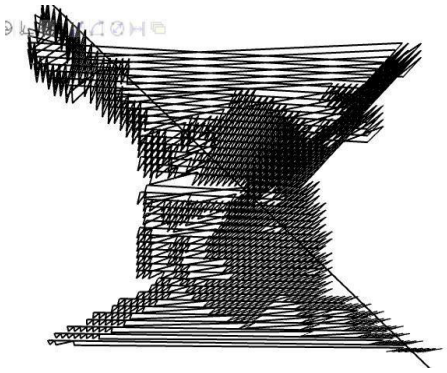
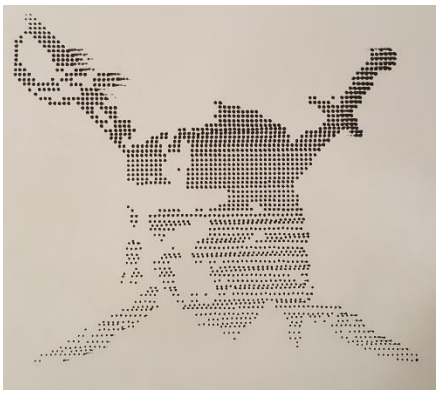
Imagen adquirida (800x600)	Imagen transformada
	
Simulación RobotStudio (100x66)	**Resultado obtenido con IRB120**
	

TABLA 5.3 Resultado dibujo por puntos piratas

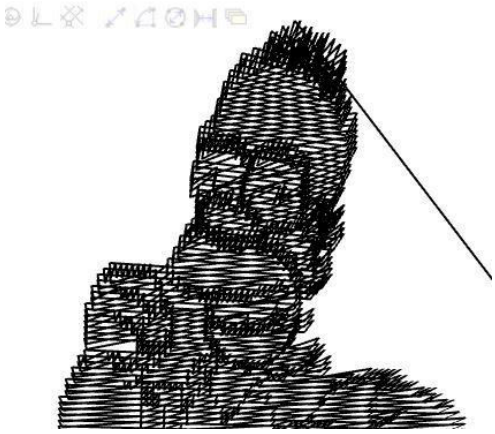
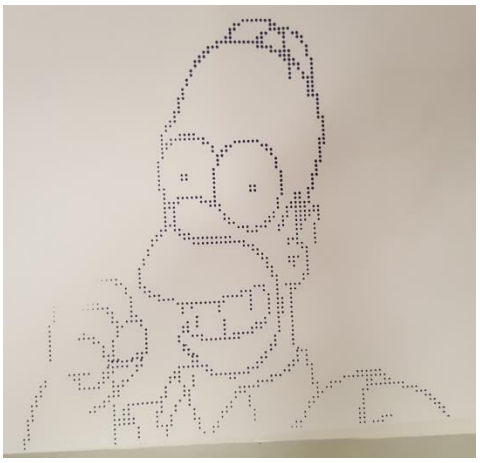
Imagen adquirida (1200x675)	Imagen transformada
	
Simulación RobotStudio (100x75)	Resultado obtenido con IRB120
	

TABLA 5.4 Resultado dibujo por puntos Homer

5.2.2.2. Aplicación dibujo por líneas

Del mismo modo que ocurre en la aplicación por puntos, a partir de los píxeles que forman parte del segmento objeto se va a establecer un dibujo mediante líneas horizontales.

- **Almacenamiento**

Mediante un algoritmo desarrollado en el proyecto anterior, que se ha mantenido en el presente, se establece cuándo existe una fila de píxeles pertenecientes al objeto y, por tanto, puede trazarse una línea para su dibujo. En el siguiente esquema se explica la metodología:

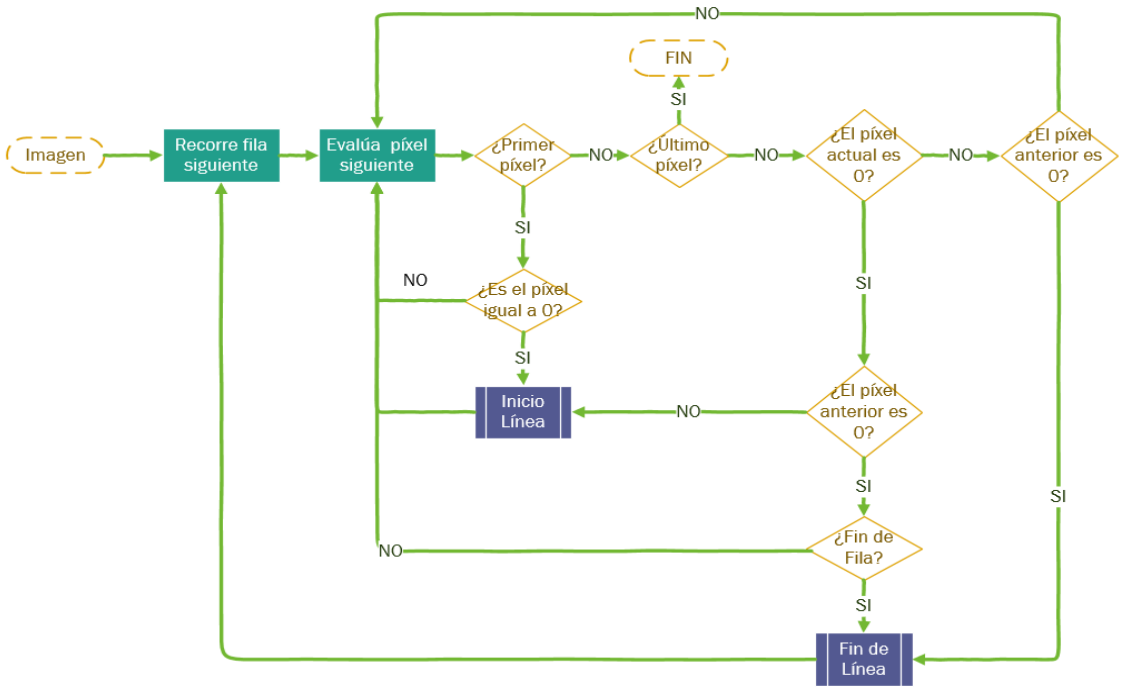


FIGURA 5.4 Almacenamiento información por líneas

Una vez almacenadas las líneas de píxeles del segmento objeto, tienen que convertirse en puntos de dibujo para que el robot pueda reproducir la trayectoria. El proyecto anterior proponía la misma solución que para la aplicación de dibujo por puntos: crear una plantilla de píxeles con tantos puntos como píxeles totales tuviera la imagen.

De nuevo, en este proyecto se ha considerado poco eficiente y por ello se ha optado por guardar únicamente las posiciones correspondientes a los puntos de dibujo y reposo de las líneas almacenadas. El proceso será el mismo que en la aplicación de dibujo, pero en este caso, cada línea estará formada por dos pares de puntos (reposo inicial – inicio dibujo - fin dibujo – reposo final).

```
Punto1=Matriz_Lineas(1,:,i);
Punto2=Matriz_Lineas(2,:,i);

L3_ini=Punto1(1)*Papel; %Ancho
L4_ini=Punto1(2)*Papel; %Alto

L3_fin=Punto2(1)*Papel; %Ancho
L4_fin=Punto2(2)*Papel; %Alto

Tbe_ini=transl(X,Y,Z)*troty(90,'deg')*ROTACION*transl(0,-L3_ini,0)*
        transl(L4_ini,0,0);
Tbe_fin=transl(X,Y,Z)*troty(90,'deg')*ROTACION*transl(0,-L3_fin,0)*
        transl(L4_fin,0,0);
```

La matriz donde se encuentran guardadas las líneas contiene 2 filas, y cada fila es un par de puntos (reposo-dibujo), por tanto, la solución más evidente para resolver este problema es crear 2 pares de variables *L3* y *L4* (desplazamiento horizontal y vertical respectivamente) de modo que uno se corresponda con el par de puntos de inicio de línea (*L3_ini* y *L4_ini*) y el otro con el de fin de línea (*L3_fin* y *L4_fin*). Del mismo modo la posición de los puntos de inicio y fin de la línea está determinada por la variable *Papel* definida por el usuario.

- **Envío**

La forma de enviar las líneas es idéntica a la de enviar puntos, solo que, en este caso, por cada línea se evalúan 2 pares de puntos, por lo que el máximo número de puntos enviados simultáneamente en este caso se limita a 32 (4 puntos por línea) para poder enviar adecuadamente en cada ocasión líneas completas sin que existan pérdidas de información. Por tanto, como las modificaciones son mínimas respecto al caso de envío de puntos, sea decidido no incluir el código a continuación, aunque siempre puede consultarse en la sección de Planos del presente libro.

- **Resultados**

En este punto se mostrarán algunos resultados obtenidos. Debe tenerse en cuenta que el resultado plasmado en el papel es idéntico al del proyecto que toma como base, pero, por cada línea trazada, en este caso, son dibujadas 3 líneas, reduciéndose el tiempo de ejecución total en una tercera parte. Debe señalarse que al ser el procedimiento de dibujo de líneas similar al de dibujo por puntos, únicamente se conseguirá una mejora de tiempo entre una línea y otra, ya que el retraso en la comunicación se produce entre dos envíos consecutivos de puntos y no afecta al propio trazo de la línea, ya que el robot se desplaza con la misma velocidad en ambos casos. Además, los resultados obtenidos en simulación muestran tanto las trayectorias de dibujo como las de reposo, pero el resultado final únicamente reflejaría el trazo del rotulador sobre el papel.

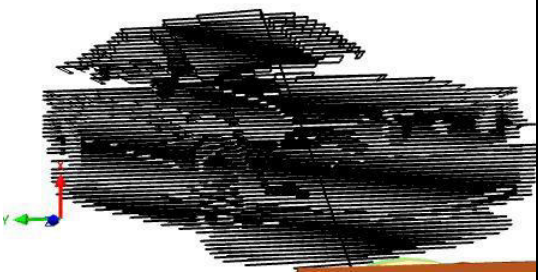
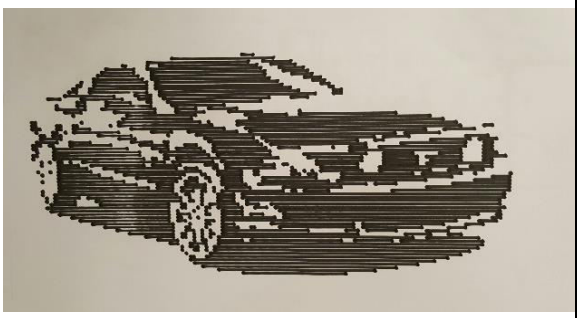
Imagen adquirida (800x464)	Imagen transformada
	
Simulación RobotStudio (133x100)	**Resultado con IRB120 (133x100)**
	

TABLA 5.5 Resultado dibujo por líneas coche

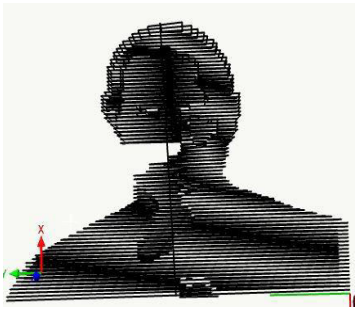
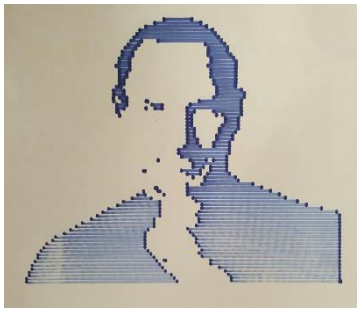
Imagen adquirida (500x432)	Imagen transformada (100x100)
	
Simulación RobotStudio	**Resultado obtenido con IRB120**
	

TABLA 5.6 Resultado dibujo por líneas retrato

5.2.2.3. Aplicación dibujo tipo pizarra

Esta aplicación va a consistir en dibujar por medio del ratón en la pantalla del ordenador enviando los puntos al robot para que pueda reproducir el dibujo realizado. Es la base para la realización de la aplicación del Phantom, por ello va a incluir puntos en común.

- **Adquisición**

El proceso de adquisición es la parte fundamental de esta aplicación y se ha mantenido del proyecto anterior. Se basa en utilizar la función *ginput* de MATLAB dentro de un bucle, lo que permite mediante el ratón *clickar* en puntos sobre unos ejes y obtener el valor de las coordenadas, las cuales serán guardadas generando líneas de un punto al siguiente.

La misma función permite además leer la tecla o botón que se ha pulsado en el ordenador (código ASCII), lo que es utilizado para establecer la tecla *n* como indicador de que se va a dibujar una línea nueva (levantar el rotulador del papel), y la tecla *s* para determinar el fin del dibujo.

Debido a que este proceso sigue la misma estructura descrita en el apartado de la aplicación de Phantom, no será expuesto su código para no resultar repetitivo.

- **Almacenamiento y envío**

Mediante una matriz, cada vez que se haga un *click* o se pulse la tecla *n* se guardarán las coordenadas del punto que será enviado al robot, bien para realizar un trazo sobre el papel o bien para levantar el rotulador.

Para su posterior representación, de nuevo se había optado en el proyecto anterior por la creación de un panel con tantos píxeles como dimensiones tuviera la hoja de dibujo. Si en las aplicaciones anteriores, parecía resultar innecesario crear puntos inutilizados, en esta aplicación cobra más sentido aún el crear únicamente los puntos de dibujo, puesto que el tipo de dibujos realizados en la mayoría de los casos contendrá pocas posiciones de trazo.

El método de envío de trayectorias es idéntico al desarrollado en la aplicación de dibujo con el Phantom, donde encontraremos las mismas condiciones evaluadas en su respectivo apartado, por lo que no se entrará en más detalle.

- **Resultados:**

En este punto se mostrarán algunos resultados obtenidos. Debe tenerse en cuenta que el resultado plasmado en el papel es idéntico al del proyecto que toma como base, pero, por cada punto de dibujo, son realizados 4, reduciendo el tiempo de ejecución total en una cuarta parte. Además, los resultados obtenidos en simulación muestran tanto las trayectorias de dibujo como las de reposo, pero el resultado final únicamente reflejaría el trazo del rotulador sobre el papel.

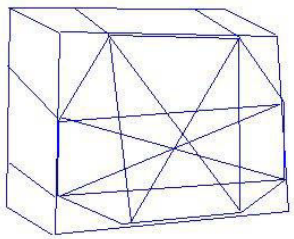
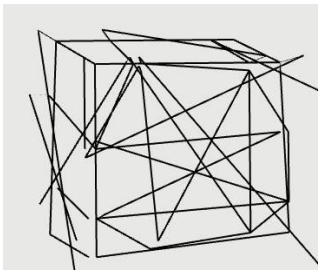
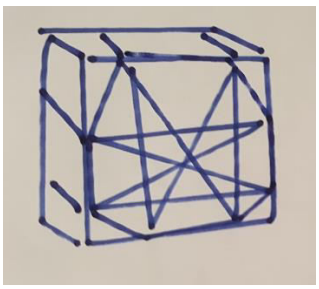
Imagen dibujada	Simulación RobotStudio	Resultado obtenido con IRB120
		

TABLA 5.7 Resultado dibujo pizarra forma geométrica

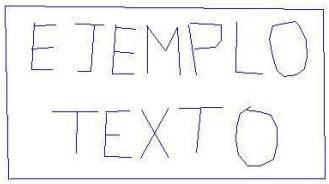
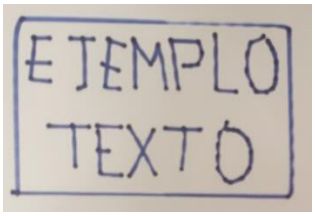
Texto escrito	Simulación RobotStudio	Resultado obtenido con IRB120
		

TABLA 5.8 Resultado dibujo pizarra texto

6. DESARROLLO DE LA INTERFAZ GRÁFICA

Una vez desarrolladas las aplicaciones, se empleará la herramienta GUIDE de MATLAB para la creación de una interfaz que permita el control de forma visual de las funcionalidades desarrolladas mediante el código.

Como ya se ha señalado en varias ocasiones, una de las partes de este proyecto consiste en la mejora del trabajo realizado en [1], y debido a que en dicho trabajo se emplea la misma herramienta para desarrollar la interfaz, se va a tomar ésta como base, se llevarán a cabo las modificaciones que se estimen convenientes y se incluirá la aplicación desarrollada con el Phantom.

6.1. Interfaz general

Partimos de una interfaz que contiene todos los ajustes y funciones necesarias para llevar a cabo las aplicaciones de dibujo por puntos y por líneas, y el reconocimiento y escritura de textos. Para la aplicación de dibujo mediante pizarra se incluye una nueva interfaz independiente a la que se puede acceder desde la primera.

El primer cambio que podemos apreciar en la nueva interfaz es su estética: se ha optado por una reordenación de los elementos que se ha considerado más intuitiva, como podemos apreciar en la imagen.

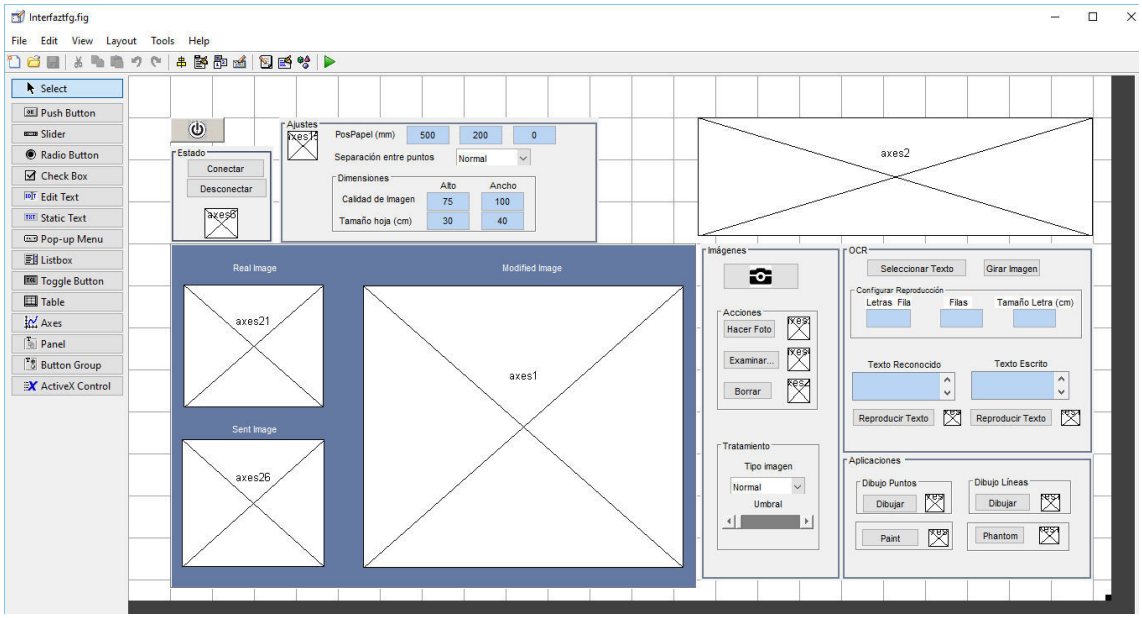


FIGURA 6.1 Interfaz gráfica general

A continuación, se van a mencionar los cambios introducidos sobre la funcionalidad:

- **Estado de conexión:**

El recuadro de la esquina superior izquierda cuenta con los botones *Conectar* y *Desconectar*. Cuando vamos a realizar la conexión se llama a la función *Estado_Conectado_Callback*, que incluye el siguiente código:

```
confirmacion = questdlg('¿Desea conectar con la estación real o la
simulación RobotStudio?','Conexión robot','Robot','RobotStudio','Robot');
switch confirmacion
    case 'Robot'
        robot=irb120('172.29.28.185',1024); %Estación real
    case 'RobotStudio'
        robot=irb120('127.0.0.1',1024); %Mismo ordenador
end

robot.connect;

ventana = waitbar(0,'Estableciendo conexión. Espere...');
steps = 1000;
for step = 1:steps
    waitbar(step / steps)
end
close(ventana)

msgbox('Éxito en la conexión');
guidata(hObject, handles);
```

Mediante la función *questdlg* aparecerá por pantalla una ventana emergente que nos dará la opción de conectarnos al simulador o al robot real. Se ha decidido incluir esta opción para realizar una conexión rápida cuando tengamos que efectuar la ejecución de un modo u otro, y así evitar tener que cambiar el código del programa.

Además, tras la conexión, mediante la función *waitbar* aparece una barra de carga en pantalla que se cierra al llegar al 100%, momento en el que el robot ha alcanzado la posición de reposo tras la conexión.

- **Dimensiones de dibujo:**

En el recuadro de *Ajustes*, dentro de *Dimensiones*, encontramos *Calidad de imagen* y *Tamaño de hoja*, que incluyen recuadros de texto editable por el usuario.

El usuario podrá modificar los recuadros correspondientes a *Tamaño de hoja* y cambiarán simultáneamente los correspondientes a *Calidad de imagen*. Esto es posible debido a que el tamaño de cada punto (píxel) en la hoja tiene un tamaño definido por *Separación de puntos*, y con las siguientes líneas de código conseguimos dibujar el máximo número de puntos en una hoja en función de sus dimensiones y así obtenemos siempre la máxima calidad de dibujo.

```
alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel; pix_a=floor(pix_a);
pix_l=ancho_hoja/Papel; pix_l=floor(pix_l);

set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);
```

Las variables *alto_hoja* y *ancho_hoja*, son las correspondientes a los recuadros que permiten establecer las dimensiones del papel, mientras que *alto* y *ancho*, son las correspondientes al número de píxeles.

- **Gráficos:**

En la anterior interfaz nos encontrábamos con un único gráfico en el que se mostraba la imagen sobre la que se estaba aplicando un tratamiento según el nivel de umbral. Sin embargo, esa imagen mostraba el tratamiento aplicado sobre las dimensiones originales, por lo que, al redimensionarla, los resultados obtenidos por el robot variaban respecto a lo que se podía esperar.

Por este motivo se ha estimado oportuno establecer tres gráficos simultáneos:

- *Real Image*: muestra la imagen original con sus dimensiones.
- *Modified Image*: este gráfico se corresponde con el que aparecía en la antigua interfaz, donde iba modificándose la imagen en función del tratamiento aplicado.
- *Sent Image*: muestra los píxeles de dibujo que serán enviados al robot. Al modificarse al mismo tiempo que *Modified Image*, permite ver el resultado que se va a obtener y en función de ello poder cambiar la separación entre pixeles o las dimensiones de la hoja.

- **Borrar gráficos:**

Cuando en la anterior interfaz se quería cargar una imagen nueva para ser enviada al robot, se sobrescribía a la anterior, lo que no permitía que se mostrara de forma adecuada. Para solventar esto, se ha creado el *Push Button* "Borrar", que crea la función *QuitaImagen_Callback,* que contiene el siguiente código:

```
axes(handles.axes1)
cla reset;
axes(handles.axes21)
cla reset;
axes(handles.axes26)
cla reset;
```

La función *cla* (*Clear Current Axes*) permite eliminar los objetos gráficos de los ejes actuales, por ese motivo se establecen como ejes cada uno de los tres que hemos visto anteriormente, y se realiza la función, posibilitando una nueva adquisición sin problema alguno.

6.2. *Interfaces secundarias*

Dentro de la interfaz general podemos encontrar dos botones que nos permiten entrar a la interfaz de la aplicación de tipo pizarra, y a la aplicación de dibujo con el Phantom, respectivamente.

A la interfaz de la aplicación tipo pizarra se le ha añadido el mismo panel de ajustes que veíamos en la interfaz general y que nos permite ajustar el dibujo al tamaño deseado por el usuario. Además, se ha modificado la función para borrar el gráfico y realizar un nuevo dibujo, implementándolo del mismo modo que hemos visto en la interfaz general.

Por su parte, la interfaz de la aplicación del Phantom presenta un diseño similar, incluyendo las funcionalidades características para su conexión o calibración como ya se explicó en el apartado correspondiente.

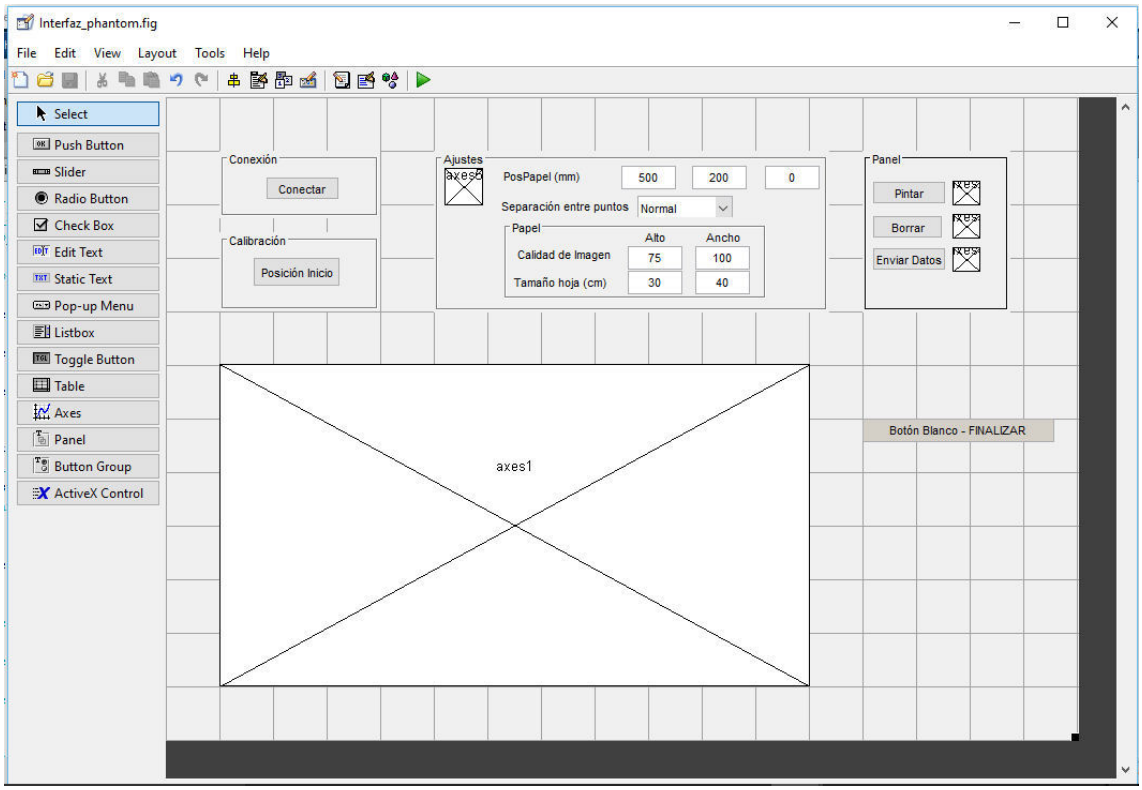


FIGURA 6.2 Interfaz gráfica de la aplicación Phantom

7. CONCLUSIONES Y TRABAJOS FUTUROS

Este proyecto se basa en el desarrollo de una aplicación mediante el dispositivo háptico Phantom Omni que consiste en la reproducción de dibujos realizados a mano alzada por el usuario, llevada cabo por el brazo robot IRB120. Esta aplicación se ha unido a las ya creadas en [1], las cuales han sido mejoradas y optimizadas, manteniendo la base de su trabajo.

Los resultados alcanzados han sido satisfactorios, ya que se ha obtenido lo que se buscaba: el desarrollo correcto de las aplicaciones mediante una comunicación fluida que se ha traducido en una mejora en los tiempos de ejecución.

En un principio, en el trabajo no se proponía la creación de un datagrama para el envío de posiciones simultáneas, ya que iba a emplearse un socket ya implementado en otro proyecto para la evitación de obstáculos, sin embargo, esto no fue posible debido a que este proyecto necesita realizar trayectorias lineales para poder mantener el rotulador sobre el papel al realizar líneas, y el proyecto ejemplo estaba basado en movimientos articulares que provocaban trayectorias imposibles.

Con todo esto, el socket creado para establecer la comunicación aprovecha al máximo su capacidad para el envío de puntos simultáneos, con lo que podemos determinar que se ha conseguido la máxima fluidez posible mediante este método de comunicación para la implementación de las aplicaciones expuestas.

La forma en la que ha sido planteado el proyecto ha permitido que únicamente utilizáramos las funciones del Phantom que determinaban su posición en el espacio, ya

que todas las posiciones se encontraban en el plano o encima de él, pero, sin embargo, también permite una lectura de la orientación de sus ejes, lo que puede resultar útil en la realización de aplicaciones futuras en el envío de trayectorias.

Del mismo modo, como se ha explicado, este dispositivo háptico permite la retroalimentación de fuerzas de manera que el usuario que sostiene el lápiz puede tener la sensación de estar tocando un objeto virtualmente. Esta característica no se ha implementado debido a que el objetivo del proyecto era controlar el brazo robot IRB120 mediante este dispositivo, pero podría resultar de gran interés la implementación de este tipo de funciones para controlar el Phantom desde un programa externo como podría ser MATLAB.

Además, la librería HAPTIK incluye la posibilidad de trabajar en MATLAB Simulink ya que incluye *S-functions* en el archivo *simulink.dll* que determinan la información del dispositivo y el tiempo de ejecución para poder sincronizarse. Por este motivo, considero un posible trabajo futuro podría ser el estudio de este archivo y el posterior desarrollo de una interfaz mediante Simulink.

En general, cabe destacar que al no existir precedentes en proyectos de la Universidad de Alcalá que utilicen el dispositivo háptico Phantom, es seguro que podrán surgir mejoras en trabajos futuros debido a las amplias posibilidades que ofrece el Phantom Omni, y que podrán tener el presente proyecto como punto de partida.

En conclusión, el presente proyecto se ha enfocado a desarrollar una aplicación de dibujo, pero su idea puede extrapolarse al ámbito industrial modificando la herramienta de trabajo, y puede servir para otras tareas como por ejemplo la soldadura.

8. PLANOS

En este capítulo encontraremos el código de los archivos realizados para ejecutar cada una de las aplicaciones. Debido a que parte del código en el apartado de mejora de aplicaciones se va a mantener no será expuesto y se invita al lector a acudir a la bibliografía del presente proyecto.

8.1. Implementación del servidor

En este apartado se expone el código desarrollado en RAPID para establecer la conexión con el usuario.

- **IRB120_Control**

En este módulo del programa desarrollado en RobotStudio encontramos la comunicación con el cliente: tanto el envío como recepción de información mediante el socket y las correspondientes órdenes de ejecución de trayectorias del robot.

```
MODULE IRB120_Control

VAR num      receivedData{1024};

VAR jointtarget axis_pos := [ [0, 0, 0, 0, 0, 0], [ 0, 9E9, 9E9, 9E9, 9E9, 9E9] ];
VAR jointtarget jtar;
VAR orient effector_orient;
VAR robtarget Target_xyz;
VAR robjoint ax;
VAR num axMatrix{6};
VAR num rotMatrix{3};
VAR num posMatrix{3};
VAR num limitMatrix{2};
VAR num velocityMatrix{2};
VAR num n;
VAR bool confFirstError;
VAR num cnfNum;

VAR  num contador:=0;
VAR num transmission:=0;

  PROC DataProcessingAndMovement()
    WHILE connectionStatus DO

      SocketReceive socketClient \Data:=receivedData \ReadNoOfBytes:=1024 \Time:= WAIT_MAX;
!Receive data from the socket
      stringHandler(receivedData);  !Handle the received data string

    ENDWHILE
   ERROR
    IF ERRNO=ERR_SOCK_CLOSED THEN
      connectionStatus := FALSE;
      SocketClose socketServer;
      WaitTime 2;
      SocketClose socketClient;
      WaitTime 2;
      main;
    ENDIF
  ENDPROC


  PROC stringHandler (num Data2Process{*})

    n := 0;
    Target_xyz := CRobT(\Tool:=tool \WObj:=wobj0);

    TEST Data2Process{1}

                              . . .

    CASE 8:
    contador:=0;
    transmission:=0;
    FOR j FROM 1 TO Data2Process{2} DO          !Número de pos y rot envíadas
      Incr transmission;
      FOR i FROM 3+contador TO 7+contador STEP 2 DO !Valor signo rotacion
        Incr n;
        IF Data2Process{i} = 0 THEN
          rotMatrix{n} := -Data2Process{i+1};
        ELSEIF Data2Process{i} = 1 THEN
          rotMatrix{n} := Data2Process{i+1};
        ENDIF
      ENDFOR
```

```
        n := 0;
        FOR i FROM 9+contador TO 15+contador STEP 3 DO !Valor signo posicion
            Incr n;
            IF Data2Process{i} = 0 THEN
                posMatrix{n} := -(100*Data2Process{i+1} + Data2Process{i+2});
            ELSEIF Data2Process{i} = 1 THEN
                posMatrix{n} := 100*Data2Process{i+1} + Data2Process{i+2};
            ENDIF
        ENDFOR

        effector_orient := OrientZYX(rotMatrix{1}, rotMatrix{2}, rotMatrix{3});
        Target_xyz.rot := effector_orient;
        Target_xyz.trans.x := posMatrix{1};
        Target_xyz.trans.y := posMatrix{2};
        Target_xyz.trans.z := posMatrix{3};

        !====== Security procedure (reachability) ======
        CheckReachabilityXYZ posMatrix;
        !================================================

        IF statusAVAILABLE THEN
            SingArea \Wrist;
            jtar := CalcJointT (Target_xyz, tool\WObj:=wobj0);
            ConfJ \Off;
            MoveL Target_xyz,vel,fine,tool\WObj:=wobj0;
            SendDataProcedure 1;
        ELSEIF statusAVAILABLE = FALSE THEN
            SendDataProcedure 2;
        ENDIF

        contador:=contador+15;
        n:=0;
        IF Data2Process{2}>1 THEN
            IF transmission=1 THEN
                SocketSend socketClient \Str:="Start";
            ENDIF
        ENDIF

    ENDFOR

  ENDTEST
                    . . .
ENDMODULE
```

8.2. *Implementación del cliente*

En este apartado se expondrán los diferentes archivos .m de MATLAB que han sido creados para la comunicación con el servidor y el desarrollo de las aplicaciones.

- **irb120.m**

En este archivo se crea la clase de MATLAB *irb120* mediante la cual podemos recibir los mensajes enviados por el servidor y enviar los datagramas para la ejecución de las trayectorias.

```
classdef irb120 < handle

    properties
        IP;
        port;
        conexion;
    end

    methods

                . . .

        function connect(r)
            r.conexion=tcpip(r.IP, r.port);
            set(r.conexion, 'Timeout', 2); %Para que lea cadena
            fopen(r.conexion);
            D0=uint8([0 1 0 0 2 1 2 1 0 1 0 1 0]); % lee reciveddata{1}

            fwrite(r.conexion,D0); %Enviamos el dato
            %Leemos conexión
            data = fread(r.conexion);
            string=char(data)'
        end

        function communicate(r)
            string='';
            while strcmp(string, 'Start')==0
                data = fread(r.conexion);
                string=char(data)'
            end
        end

        function socketfree(r)
            data = fread(r.conexion);
            string=char(data)'
        end

              . . .

         function TCPTrajectory(r,tcps)
       %Primera posición id; segunda n°puntos
              D8=uint8([8 length(tcps)/6]);
              posiciones=3; %Desplazamiento posiciones
              posicion=1;%Se desplaza por las pos y rot

              for i=1:(length(tcps)/6) %Para cada punto
                  tcp(1:6)=tcps(posicion:posicion+5
                  posicion=((i*6)+1); %Se actualiza

                  Rz=uint8(round(abs(tcp(1))));
                  signorz = sign (tcp(1));
                  if signorz == -1
                      sz=0;
                  else
                      sz=1;
                  end

                  Ry= uint8(round(abs(tcp(2))));
                  signory = sign (tcp(2));
                  if signory == -1
                      sy=0;
                  else
                      sy=1;
                  end
```

```
                Rx= uint8(round(abs(tcp(3))));
                signorx = sign (tcp(3));
                if signorx == -1
                    sx=0;
                else
                    sx=1;
                end

                xr=rem(abs(tcp(4)),100);
                X=floor(abs(tcp(4))/100);
                signopx = sign (tcp(4));
                if signopx == -1
                    spx=0;
                else
                    spx=1;
                end

                yr=rem(abs(tcp(5)),100);
                Y=floor(abs(tcp(5))/100);
                signopy = sign (tcp(5));
                if signopy == -1
                    spy=0;
                else
                    spy=1;
                end

                zr=rem(abs(tcp(6)),100);
                Z=floor(abs(tcp(6))/100);
                signopz = sign (tcp(6));
                if signopz == -1
                    spz=0;
                else
                    spz=1;
                end

                d8=[sz Rz sy Ry sx Rx spx X xr spy Y yr spz Z zr];

                D8(posiciones:posiciones+14)=uint8(d8);
                posiciones=(i*15)+3;
            end
            %Enviamos el dato
            fwrite(r.conexion,D8);
            pause(1);
        end
    end
end
```

- **Interfaztfg.m**

Este archivo contiene el código desarrollado en la creación de la interfaz gráfica mediante la herramienta GUIDE de MATLAB. En él se encuentran las llamadas a las funciones que permiten el desarrollo de las aplicaciones expuestas.

```
function varargout = Interfaztfg(varargin)

                . . .

% Choose default command line output for Interfaztfg
handles.output = hObject;
axes(handles.axes2)
imshow('gui_logoUAH1.jpg')
axes(handles.axes3)
imshow('gui_HacerFoto.jpg')
axes(handles.axes4)
imshow('gui_Carpeta.jpg')
axes(handles.axes5)
imshow('gui_Paint.jpg')
axes(handles.axes6)
imshow('gui_desconectado.jpg')
axes(handles.axes8)
imshow('gui_compila_escribir.jpg')
axes(handles.axes9)
imshow('gui_compila_puntos.jpg')
axes(handles.axes10)
imshow('gui_compila_lineas.jpg')
axes(handles.axes15)
imshow('gui_ajustes.jpg')
axes(handles.axes16)
imshow('gui_compila_escribir.jpg')
axes(handles.axes19)
imshow('gui_phantom.jpg')
axes(handles.axes20)
imshow('gui_goma.jpg')

axes(handles.axes1)

% Update handles structure
guidata(hObject, handles);

              . . .

% --- Executes on button press in examinar.
function examinar_Callback(hObject, eventdata, handles)
global foto1;
global roll;
global foto3;

axes(handles.axes1)
foto=imgetfile;
foto1=imread(foto);
imshow(foto1);



foto3=foto1;
axes(handles.axes21)
imshow(foto3);
roll=0;
guidata(hObject,handles)
```

```
% --- Executes on slider movement.
function umbral_Callback(hObject, eventdata, handles)
          . . .
global foto1;
global I3;
global Papel
global ancho
global alto
global alto_hoja
global ancho_hoja
global B

valor_umbral=get(handles.umbral,'Value');
se1=strel('line',2,0);
se2=strel('line',3,90);
e=get(handles.Tipo_imagen,'Value')
axes(handles.axes1)
%-----------------------------------------------
switch e
    case 1
        I2=rgb2gray(foto1);
        I3=im2bw(I2,valor_umbral);
        imshow(I3);
    case 2
        I2=rgb2gray(foto1);
        imgbw=im2bw(I2,valor_umbral);
        I3=imdilate(imgbw,[se1,se2],'full');
        imshow(I3);
    case 3
        I2=rgb2gray(foto1);
       imgbw=im2bw(I2,valor_umbral);
        I3=imerode(imgbw,[se1,se2],'full');
        imshow(I3);
end

axes(handles.axes26)

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel;
pix_a=floor(pix_a);
pix_l=ancho_hoja/Papel;
pix_l=floor(pix_l);

set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);
```

```
alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

B=imresize(I3,[alto ancho]);
imshow(B);
          . . .
% --- Executes on button press in ImagenPuntos.
function ImagenPuntos_Callback(hObject, eventdata, handles)

global foto1;
global I3;
global robot
global ancho;
global alto;
global ancho_hoja;
global alto_hoja;
global X
global Y
global Z
global Papel
global Matriz_puntos

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

Aplicacion_artistica_puntos_GUI
Panel_puntos_GUI
msgbox('Enviando datos al robot IRB120...');
Aplicacion_dibujo_puntos_GUI
msgbox('Datos enviados');

% --- Executes on button press in ImagenLineas.
function ImagenLineas_Callback(hObject, eventdata, handles)

global foto1;
global I3;
global robot
global X
global Y
global Z
global ancho
global alto
global ancho_hoja
global alto_hoja
global Papel
global Matriz_Lineas

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));
```

```
Aplicacion_artistica_lineas_GUI
Panel_Lineas_GUI
msgbox('Enviando datos al robot IRB120...');
Aplicacion_dibujo_lineas_GUI
msgbox('Datos enviados');
            . . .
% --- Executes on button press in texto.
function texto_Callback(hObject, eventdata, handles)

global n_letras_fila
global n_filas
global Letra
global reconocimiento
global robot
global X
global Y
global Z
global alto_hoja
global ancho_hoja

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

n_letras_fila=str2double(get(handles.LetrasFila,'String'));
n_filas=str2double(get(handles.filas,'String'));
Letra=str2double(get(handles.TLetra,'String'));

Panel_ocr_GUI
msgbox('Enviando datos al robot IRB120...');
Aplicacion_alfabetica_ocr_GUI;
msgbox('Datos enviados');
guidata(hObject, handles);
              . . .
% --- Executes on button press in Estado_Conectado.
function Estado_Conectado_Callback(hObject, eventdata, handles)

global robot

axes(handles.axes6)
imshow('gui_conectado.jpg')

confirmacion = questdlg('¿Desea conectar con la estación real o la
simulación RobotStudio?','Conexión
robot','Robot','RobotStudio','Robot');
switch confirmacion
    case 'Robot'
        robot=irb120('172.29.28.185',1024); %Estación real
    case 'RobotStudio'
%         robot=irb120('172.22.7.102',1024);
        robot=irb120('127.0.0.1',1024); %Mismo ordenador
end

robot.connect;
```

```
ventana = waitbar(0,'Estableciendo conexión. Espere...');
steps = 1000;
for step = 1:steps
    waitbar(step / steps)
end
close(ventana)

msgbox('Éxito en la conexión');
guidata(hObject, handles);
        . . .

% --- Executes on button press in seguridad.
function seguridad_Callback(hObject, eventdata, handles)

global robot

robot.TCPTrajectory([-180 0 -180 300 0 330]);
robot.socketfree;
        . . .

% --- Executes on button press in texto2.
function texto2_Callback(hObject, eventdata, handles)

global n_letras_fila
global n_filas
global Letra
global reconocimiento
global robot
global X
global Y
global Z
global alto_hoja
global ancho_hoja

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

n_letras_fila=str2double(get(handles.LetrasFila,'String'));
n_filas=str2double(get(handles.filas,'String'));
Letra=str2double(get(handles.TLetra,'String'));
reconocimiento=get(handles.textoescrito,'String');

Panel_ocr_GUI
msgbox('Enviando datos al robot IRB120...');
Aplicacion_alfabetica_ocr_GUI;
msgbox('Datos enviados');
guidata(hObject, handles);
         . . .

function Phantom_Callback(hObject, eventdata, handles)

Interfaz_phantom
```

```
% --- Executes on button press in QuitaImagen.
function QuitaImagen_Callback(hObject, eventdata, handles)

global foto1
global foto2
global foto3
global B

axes(handles.axes1)
cla reset;
axes(handles.axes21)
cla reset;
axes(handles.axes26)
cla reset;

clear foto1;clear foto2;clear foto3;clear B

function ancho_hoja_Callback(hObject, eventdata, handles)

global ancho_hoja
global ancho
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_l=ancho_hoja/Papel;
pix_l=floor(pix_l);

set(handles.ancho,'String',pix_l);
ancho=str2double(get(handles.ancho,'String'));
           . . .

function alto_hoja_Callback(hObject, eventdata, handles)

global alto_hoja
global alto

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end
```

```
pix_a=alto_hoja/Papel;
pix_a=floor(pix_a);

set(handles.alto,'String',pix_a);
alto=str2double(get(handles.alto,'String'));
     . . .
```

- **Aplicación_alfabetica_ocr_GUI.m**

En este archivo se asignará el código ASCII de cada carácter del texto escrito o reconocido a su letra o número correspondiente, para después enviar la trayectoria de cada carácter al robot simultáneamente.

```
cadena=reconocimiento;
orden=1;
space=0;

for j=1:1:length(cadena) %Para cada letra
    a=0;

    if abs(cadena(j))==97 |abs(cadena(j))==65
        numero=letraa(:,:,orden);
        orden=orden+1;
            . . .
    elseif abs(cadena(j))==122|abs(cadena(j))==90
        numero=letraz(:,:,orden);
        orden=orden+1;
    elseif abs(cadena(j))==32
        orden=orden+1;
        space=1;

    % ---------------------------------

    elseif abs(cadena(j))==48
        numero=cero(:,:,orden);
        orden=orden+1;
            . . .
    elseif abs(cadena(j))==57
        numero=nueve(:,:,orden);
        orden=orden+1;
   % ---------------------------------
    else
        space=1;
        orden=orden+1;
    end

    posi=1;
    tcp=[];

    for int=1:1:((length(numero)/4)) %cada letra abarca matrices 4x4
        Tpos=numero(1:4,1+a:4+a); %recorre cada vez 4 columnas (punto)
        pos=transl(Tpos); pos=pos*1000;  %posiciones a milímetros
        rot=tr2rpy(Tpos); rot=rot*180/pi; %ángulos a grados
        tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)];

        posi=posi+1;
        a=a+4;
    end
```

```
    cont=1;
    for i=1:(posi-1) %Número de puntos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
        cont=(i*6)+1;
    end

    %Se mantiene a la espera hasta enviar siguiente trayectoria
    if space==0
        robot.TCPTrajectory(tcps)
        robot.communicate
    else
        space=0;
        robot.socketfree;
    end

    %Borrar en cada iteración
    numero=[];
    tcp=[];
    tcps=[];
end
robot.socketfree;
```

- **Panel_puntos_GUI.m**

En este archivo se crea el panel con los píxeles que posteriormente serán enviados al robot.

```
L3=0; %ancho
L4=0; %alto
n_puntos=size(Matriz_puntos);

for i=1:1:n_puntos(3)
    contador=i
    angle=90;
    ROTACION=troty(angle,'deg');

    Puntos=Matriz_puntos(:,:,i);
    L3=Puntos(2)*Papel; %Ancho
    L4=Puntos(1)*Papel; %Alto

    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);

    hold on

    x1=0;x2=0;
    y1=0;y2=0;
    z1=-0.005;z2=-0.05; %Distancia seguridad

    P1=[x1 y1 z1 1]'; T1(:,i)=(Tbe)*P1;
    P2=[x2 y2 z2 1]'; T2(:,i)=(Tbe)*P2;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    Tp1(:,:,i)=[-1 0 0 u1;0 1 0 v1;0 0 -1 w1;0 0 0 1];
    u2=T2(1,i);v2=T2(2,i);w2=T2(3,i);
    Tp2(:,:,i)=[-1 0 0 u2;0 1 0 v2;0 0 -1 w2;0 0 0 1];

    hold on;
    Punto(:,:,i)=[Tp2(:,:,i) Tp1(:,:,i)];
end
```

- **Aplicacion_dibujo_puntos_GUI.m**

En este archivo se leerán cada uno de los puntos guardados en el panel para después enviar el conjunto de puntos al robot simultáneamente.

```
n_puntos=size(Matriz_puntos);
pos=1;
orden=1;

for j=1:1:n_puntos(3) %La tercera dimensión indica el número de puntos
    a=0;
    orden=orden+1;

    pos_img=j;
    numero=Punto(:,:,pos_img);

    for int=1:1:((length(numero)/4))% 2 posiciones por punto
        Tpos(:,:,pos)=numero(1:4,1+a:4+a);

        position=transl(Tpos(:,:,pos)); position=position*1000
        rotation=tr2rpy(Tpos(:,:,pos)); rotation=rotation*180/pi

        tcp(pos,:)=[rotation(3) rotation(2) rotation(1) position(1)
                    position(2) position(3)]

        a=a+4;
        pos=pos+1;
    end

    if pos>34 %17 pares de puntos máximo envío
        cont=1;
        for i=1:(pos-1) %Número de puntos
            tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
            cont=(i*6)+1;
        end

        robot.TCPTrajectory(tcps)%Dibuja los puntos

        %Se mantiene a la espera hasta enviar siguiente trayectoria
        if orden<=n_puntos(3)
            robot.communicate
        else
            robot.socketfree;
        end

        pos=1;
        %Borrar en cada iteración
        tcps=[];
    end
end

%Cuando quedan menos de 34 puntos para finalizar
if pos>1
    tcps=[];
    cont=1;
    for i=1:(pos-1) %Número de puntos de Tpos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca en una fila
        cont=(i*6)+1;
    end
    robot.TCPTrajectory(tcps)%Dibuja los puntos
    robot.socketfree; %Libera el socket
end
```

- **Panel_Lineas_GUI.m**

En este archivo se crea el panel con las líneas de píxeles que posteriormente serán enviadas al robot.

```
L3_ini=0; L3_fin=0; %ancho
L4_ini=0; L4_fin=0; %alto

n_puntos=size(Matriz_Lineas)

for i=1:1:n_puntos(3)
    contador=i
    angle=90;
    ROTACION=troty(angle,'deg');

    Punto1=Matriz_Lineas(1,:,i);
    Punto2=Matriz_Lineas(2,:,i);

    L3_ini=Punto1(1)*Papel;  L3_fin=Punto2(1)*Papel; %Ancho
    L4_ini=Punto1(2)*Papel;  L4_fin=Punto2(2)*Papel; %Alto

    Tbe_ini=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
            transl(0,-L3_ini,0)*transl(L4_ini,0,0);

    Tbe_fin=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
            transl(0,-L3_fin,0)*transl(L4_fin,0,0);

    hold on

    %Definimos los puntos de nuestro minipanel%
    x1=0;x2=0;
    y1=0;y2=0;
    z1=-0.005;z2=-0.05; %Distancia seguridad

    P1=[x1 y1 z1 1]';
    T1_ini(:,i)=(Tbe_ini)*P1;
    T1_fin(:,i)=(Tbe_fin)*P1;

    P2=[x2 y2 z2 1]';
    T2_ini(:,i)=(Tbe_ini)*P2;
    T2_fin(:,i)=(Tbe_fin)*P2;

    u1_ini=T1_ini(1,i);v1_ini=T1_ini(2,i);w1_ini=T1_ini(3,i);
    Tp1_ini(:,:,i)=[-1 0 0 u1_ini;0 1 0 v1_ini;0 0 -1 w1_ini;0 0 0 1];

    u1_fin=T1_fin(1,i);v1_fin=T1_fin(2,i);w1_fin=T1_fin(3,i);
    Tp1_fin(:,:,i)=[-1 0 0 u1_fin;0 1 0 v1_fin;0 0 -1 w1_fin;0 0 0 1];

    u2_ini=T2_ini(1,i);v2_ini=T2_ini(2,i);w2_ini=T2_ini(3,i);
    Tp2_ini(:,:,i)=[-1 0 0 u2_ini;0 1 0 v2_ini;0 0 -1 w2_ini;0 0 0 1];

    u2_fin=T2_fin(1,i);v2_fin=T2_fin(2,i);w2_fin=T2_fin(3,i);
    Tp2_fin(:,:,i)=[-1 0 0 u2_fin;0 1 0 v2_fin;0 0 -1 w2_fin;0 0 0 1];

    hold on;
    %plot3(u1,v1,w1,'ob',u2,v2,w2,'ob');

    Punto_inicio(:,:,i)=[Tp2_ini(:,:,i) Tp1_ini(:,:,i)];
    Punto_fin(:,:,i)=[Tp1_fin(:,:,i) Tp2_fin(:,:,i)];
end
```

- **Aplicacion_dibujo_lineas_GUI.m**

En este archivo se leerán cada una de las líneas guardadas en el panel para después enviar el conjunto de posiciones al robot simultáneamente.

```
n_lineas=size(Matriz_Lineas);
posi=1;
orden=1;

for j=1:1:n_lineas(3)
    orden=orden+1;
    for pos_ini_fin=1:1:2 %Cada iteración da 4 puntos
        a=0;
        pos_img=j;

        if pos_ini_fin==1
            numero=Punto_inicio(:,:,pos_img); %Va a tener 2 posiciones
        elseif pos_ini_fin==2
            numero=Punto_fin(:,:,pos_img); %Va a tener 2 posiciones
        end

        for int=1:1:((length(numero)/4))
            Tpos=numero(1:4,1+a:4+a)

            pos=transl(Tpos); pos=pos*1000  %posiciones a milímetros
            rot=tr2rpy(Tpos); rot=rot*180/pi %ángulos a grados
            tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)]

            posi=posi+1;
            a=a+4;
        end
    end

    if posi>32 %8 líneas (4 puntos por línea)
        cont=1;
        for i=1:(posi-1) %Número de puntos
            tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
            cont=(i*6)+1;
        end

        robot.TCPTrajectory(tcps)%Dibuja las líneas

        if orden<=n_lineas(3)
            robot.communicate;% Espera hasta enviar siguiente trayectoria
        else
            robot.socketfree
        end

        posi=1;
        %Borrar en cada iteración
        tcps=[];
    end
end

%Cuando quedan menos de 8 líneas por dibujar (menos de 32 posiciones)
if posi>1
    tcps=[];
    cont=1;
    for i=1:(posi-1) %Número de puntos de Tpos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca en una fila
        cont=(i*6)+1;
    end
    robot.TCPTrajectory(tcps)%Dibuja los puntos
    robot.socketfree;
end
```

- **Interfaz_paint.m**

Este archivo contiene el código desarrollado en la creación de la interfaz gráfica de la aplicación tipo pizarra. En él se encuentran las llamadas a las funciones que permiten el desarrollo de la aplicación.

```
function varargout = Interfaz_paint(varargin)
            . . .
% --- Executes just before Interfaz_paint is made visible.
function Interfaz_paint_OpeningFcn(hObject, eventdata, handles, varargin)

handles.output = hObject;
axes(handles.axes2)
imshow('gui_lapiz.jpg')
axes(handles.axes3)
imshow('gui_goma.jpg')
axes(handles.axes4)
imshow('gui_compilar.jpg')
axes(handles.axes5)
imshow('gui_ajustes.jpg')
axes(handles.axes1)
% Update handles structure
guidata(hObject, handles);
            . . .
% --- Executes on button press in Pintar.
function Pintar_Callback(hObject, eventdata, handles)

global roll
global robot
global T_img
global Matriz_paint

global X
global Y
global Z
global Papel
global alto_hoja
global ancho_hoja
global alto
global ancho

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel; pix_a=floor(pix_a);
pix_l=ancho_hoja/Papel; pix_l=floor(pix_l);
```

```
set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

Paint_GUI

% --- Executes on button press in Borrar.
function Borrar_Callback(hObject, eventdata, handles)

global Matriz_paint
cla reset;
Matriz_paint=[];

% --- Executes on button press in Envio.
function Envio_Callback(hObject, eventdata, handles)

global Matriz_paint
global robot
global T_img
global X
global Y
global Z
global Papel

global alto_hoja
global ancho_hoja
global alto
global ancho

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel;  pix_a=floor(pix_a);
pix_l=ancho_hoja/Papel; pix_l=floor(pix_l);

set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

Panel_paint_GUI
msgbox('Enviando datos al robot IRB120 ...');
Aplicacion_paint_GUI
msgbox('Datos enviados');
         ...
```

```
function ancho_hoja_Callback(hObject, eventdata, handles)

global ancho_hoja

ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_l=ancho_hoja/Papel;
pix_l=floor(pix_l);

set(handles.ancho,'String',pix_l);
          . . .

function alto_hoja_Callback(hObject, eventdata, handles)

global alto_hoja

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel;
pix_a=floor(pix_a);

set(handles.alto,'String',pix_a);
        . . .
```

- **Panel_paint_GUI.m**

En este archivo se crea el panel con los píxeles adquiridos por la aplicación que posteriormente serán enviadas al robot.

```
T_img=[alto ancho] %Tamaño de la hoja
n_puntos=size(Matriz_paint);

L3=0; %ancho (hoja horizontal)
L4=0; %largo

Puntos=[];

for i=1:1:4 %Vértices zona dibujo
    if i==2 %Arriba derecha
        L3=Papel*T_img(1);
        L4=0;
    elseif i==3 %Abajo derecha
        L3=Papel*T_img(1);
        L4=Papel*T_img(2);
    elseif i==4 %Abajo izquierda
        L3=0;
        L4=Papel*T_img(2);
    end

    angle=90; ROTACION=troty(angle,'deg');

    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);
    hold on

    %Definimos los puntos de nuestro minipanel%
    x1=0;y1=0;z1=0;
    P1=[x1 y1 z1 1]';   T1(:,i)=(Tbe)*P1;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    Tp1(:,:,i)=[-1 0 0 u1;0 1 0 v1;0 0 -1 w1;0 0 0 1];

    hold on;
    plot3(u1,v1,w1,'or');
end

for i=1:1:n_puntos(3) %Puntos de dibujo
    Puntos=Matriz_paint(:,:,i)
    L3=Puntos(1)*Papel; %Coordenada x (30cm máximo)
    L4=Puntos(2)*Papel; %Coordenada y (m)

    angle=90; ROTACION=troty(angle,'deg');

    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);

    hold on

    x1=0;x2=0;
    y1=0;y2=0;
    z1=-0.005;z2=-0.05;

    P1=[x1 y1 z1 1]';    T1(:,i)=(Tbe)*P1;
    P2=[x2 y2 z2 1]';    T2(:,i)=(Tbe)*P2;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    Tp1(:,:,i)=[-1 0 0 u1;0 1 0 v1;0 0 -1 w1;0 0 0 1];
    u2=T2(1,i);v2=T2(2,i);w2=T2(3,i);
    Tp2(:,:,i)=[-1 0 0 u2;0 1 0 v2;0 0 -1 w2;0 0 0 1];

    hold on;
%     plot3(u1,v1,w1,'ob',u2,v2,w2,'ob');

    Pintar(:,:,i)=[Tp1(:,:,i)];
    No_pintar(:,:,i)=[Tp2(:,:,i)];
end
```

- **Aplicacion_paint_GUI.m**

En este archivo se leerán cada uno de los puntos dibujados en el panel para después enviar el conjunto de posiciones al robot simultáneamente.

```
n_lineas=size(Matriz_paint);
posi=1;
orden=1;
numero=[];

for j=1:1:n_lineas(3)%Recorremos la Matriz_paint
    orden=orden+1;
    a=0;
    if j==1 %Primera iteracion
        posicion=Matriz_paint(:,:,j);
        pos_img=j;

        if posicion(3)==1 %*click*
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);
        elseif(posicion(3)==110 | poscion(3)==78)%Letra N
            numero=No_pintar(:,:,pos_img);
        end

    else
        posicion=Matriz_paint(:,:,j);
        posicion_anterior=Matriz_paint(:,:,j-1);
        pos_img_ant=j-1;
        pos_img=j;

        if (posicion(3)==1 & posicion_anterior(3)==1) %Dibuja línea
            numero=Pintar(:,:,pos_img);

        elseif (posicion(3)==110 | posicion(3)==78) %Levantar rotulador
            numero=No_pintar(:,:,pos_img_ant);

        elseif (posicion(3)==1) &  (posicion_anterior(3)==110 |
                posicion_anterior(3)==78) %Bajar rotulador
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);

        elseif (posicion(3)==115 | posicion(3)==83)
             numero=No_pintar(:,:,pos_img);
        end
    end

    for int=1:1:((length(numero)/4))
         Tpos=numero(1:4,1+a:4+a)
         pos=transl(Tpos); pos=pos*1000  %posiciones a milímetros
         rot=tr2rpy(Tpos); rot=rot*180/pi %ángulos a grados
         tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)]

         posi=posi+1;
         a=a+4;
    end

    if posi>=34 %34puntos máximo
        cont=1;
        for i=1:(posi-1) %Número de puntos
            tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
            cont=(i*6)+1;
        end

        robot.TCPTrajectory(tcps)%Dibuja las líneas
```

```
        if orden<=n_lineas(3)
            robot.communicate;  %Espera hasta siguiente trayectoria
        else
            robot.socketfree;
        end

        posi=1;

        %Borrar en cada iteración
        tcps=[];
        numero=[];%No todos tienen la misma dimensión
    end
end

%Cuando quedan menos de 34 puntos
if posi>1
    tcps=[];
    cont=1;
    for i=1:(posi-1) %Número de puntos de Tpos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca las q en una fila
        cont=(i*6)+1;
    end
    robot.TCPTrajectory(tcps)%Dibuja los puntos
    robot.socketfree;
end
```

- **Interfaz_phantom.m**

Este archivo contiene el código desarrollado en la creación de la interfaz gráfica de la aplicación con el Phantom. En él se encuentran las llamadas a las funciones que permiten el desarrollo de la aplicación.

```
function varargout = Interfaz_phantom(varargin)
          . . .
% --- Executes just before Interfaz_phantom is made visible.
function Interfaz_phantom_OpeningFcn(hObject, eventdata, handles, varargin)
% Choose default command line output for Interfaz_phantom
handles.output = hObject;

axes(handles.axes2)
imshow('gui_lapiz.jpg')
axes(handles.axes3)
imshow('gui_goma.jpg')
axes(handles.axes4)
imshow('gui_compilar.jpg')
axes(handles.axes5)
imshow('gui_ajustes.jpg')
axes(handles.axes1)

% Update handles structure
guidata(hObject, handles);
       . . .
% --- Executes on button press in CalibrarPosicion.
function CalibrarPosicion_Callback(hObject, eventdata, handles)

global xcalib
global ycalib
global zcalib
global h
```

```
msgbox('Pulsa el botón oscuro para calibrar...');
while(1)
    button = read_button(h)
    if button==1
        break
    end
end

pos_ph=read_position(h); %valores experimentales en mm
xorigen=pos_ph(3); xcalib=-xorigen
yorigen=pos_ph(1); ycalib=-yorigen
zorigen=pos_ph(2); zcalib=-zorigen
msgbox('Calibrado correctamente');
      . . .
% --- Executes on button press in Dibuja_Phantom.
function Dibuja_Phantom_Callback(hObject, eventdata, handles)

global robot
global T_img
global Matriz_phantom
global n_puntos

global h
global xcalib
global ycalib
global zcalib

global X
global Y
global Z
global Papel
global alto_hoja
global ancho_hoja
global alto
global ancho

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;


v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel; pix_a=floor(pix_a);
pix_l=ancho_hoja/Papel; pix_l=floor(pix_l);

set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

Dibujo_Phantom_GUI
```

```
% --- Executes on button press in Borra_Phantom.
function Borra_Phantom_Callback(hObject, eventdata, handles)

global Matriz_phantom

cla reset;
Matriz_phantom=[];

% --- Executes on button press in Compila_Phantom.
function Compila_Phantom_Callback(hObject, eventdata, handles)

global Matriz_phantom
global n_puntos
global robot
global T_img

global X
global Y
global Z
global Papel
global alto_hoja
global ancho_hoja
global alto
global ancho

X=str2double(get(handles.X,'String')); X=X*10^-3;
Y=str2double(get(handles.Y,'String')); Y=Y*10^-3;
Z=str2double(get(handles.Z,'String')); Z=Z*10^-3;

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;
ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel;
pix_a=floor(pix_a);

pix_l=ancho_hoja/Papel;
pix_l=floor(pix_l);

set(handles.alto,'String',pix_a);
set(handles.ancho,'String',pix_l);

alto=str2double(get(handles.alto,'String'));
ancho=str2double(get(handles.ancho,'String'));

Panel_phantom_GUI
msgbox('Enviando datos al robot IRB120 ...');
Aplicacion_phantom_GUI
msgbox('Datos enviados');
```

```
% --- Executes on button press in Conecta_Phantom.
function Conecta_Phantom_Callback(hObject, eventdata, handles)

global h

haptikdevice_list
h=haptikdevice
    . . .
function ancho_hoja_Callback(hObject, eventdata, handles)

global ancho_hoja

ancho_hoja=str2double(get(handles.ancho_hoja,'String'));
ancho_hoja=ancho_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_l=ancho_hoja/Papel; pix_l=floor(pix_l);

set(handles.ancho,'String',pix_l);
       . . .
function alto_Callback(hObject, eventdata, handles)

global alto_hoja

alto_hoja=str2double(get(handles.alto_hoja,'String'));
alto_hoja=alto_hoja*10^-2;

v=get(handles.Tamano_papel,'Value') %Separación entre puntos
switch v
    case 1
        Papel=3*10^-3; %30 cm hoja entre tamaño pixeles
    case 2
        Papel=4*10^-3;
    case 3
        Papel=2*10^-3;
end

pix_a=alto_hoja/Papel; pix_a=floor(pix_a);

set(handles.alto,'String',pix_a);
```

- **Dibujo_Phantom_GUI.m**

Este archivo contiene el código necesario para la adquisición de la posición del Phantom y su reproducción simultánea sobre un gráfico.

```
T_img=[alto ancho] %80 120; 110 200

axis([0,T_img(1),0,T_img(2)])%Indicamos el tamaño de la imagen
axis on;hold on;

a=[ ]; b=[ ];
a_ant=[]; b_ant=[];

newline=0;
n_puntos=0
Matriz_phantom=[];
camroll(-90)

while(1)
    button = read_button(h)
    if button==1
        break
    end
end

while(1)
    while(1)
        pos_ph=read_position(h);
        x=pos_ph(3)+xcalib; x=x*0.75;
        y=pos_ph(1)+ycalib; y=y*0.75;
        z=pos_ph(2)+zcalib;

        button = read_button(h);
        %get button status. 1 negro, 2 blanco, 3 ambos

        if z<3 %representacion del la linea
            newline=1;
            n_puntos=n_puntos+1;
            pintar=1;

            a=[a_ant,x]; %Dibujo lineas: pto anterior - pto. actual
            b=[b_ant,y];

            axes(handles.axes1)% para que dibuje a la vez
            plot(a,b)

            hold on;

            a_ant=x; b_ant=y;

             Matriz_phantom(:,:,n_puntos)=floor([y,x,button,pintar])

            if n_puntos>1 %Para evitar sobreescribir el mismo punto
                if (Matriz_phantom(:,1,n_puntos)==
                    Matriz_phantom(:,1,n_puntos-1))&
                   (Matriz_phantom(:,2,n_puntos)==
                    Matriz_phantom(:,2,n_puntos-1))

                    n_puntos=n_puntos-1;
                  break
                end
            end

            if (x>T_img(1) | x<0 | y>T_img(2) | y<0)
                x=0;y=0;
                n_puntos=n_puntos-1;
                break
            end
            break
        end
```

```
        if (z>=3) & (newline==1) %nueva linea  (botón blanco)
            newline=0;
            n_puntos=n_puntos+1;
            pintar=0;

            a=[ ]; b=[ ];
            a_ant=[]; b_ant=[];

            Matriz_phantom(:,:,n_puntos)=floor([y,x,button,pintar])
            break;
        end

        if button==2 %fin del dibujo pulsando blanco
            newline=0;
            n_puntos=n_puntos+1;
            pintar=0;

            a=[ ]; b=[ ];
            a_ant=[]; b_ant=[];

            Matriz_phantom(:,:,n_puntos)=floor([y,x,button, pintar])
            break;
        end

    end
    if button==2 %fin del dibujo pulsando blanco
        break;
    end
end
hold off
```

- **Panel_phantom_GUI.m**

En este archivo se crea el panel con los píxeles adquiridos por la aplicación que posteriormente serán enviadas al robot.

```
T_img=[alto ancho] %Tamaño de la hoja
L3=0; %ancho (hoja horizontal)
L4=0; %alto
n_puntos=size(Matriz_phantom);

Puntos=[];

for i=1:1:4 %Vértices zona dibujo

    if i==2 %Arriba derecha
        L3=Papel*T_img(1);
        L4=0;
    elseif i==3 %Abajo derecha
        L3=Papel*T_img(1);
        L4=Papel*T_img(2);
    elseif i==4 %Abajo izquierda
        L3=0;%30cm de la hoja entre 100 pixeles = 0.003 (ancho)
        L4=Papel*T_img(2);
    end

    angle=90;
    ROTACION=troty(angle,'deg');
    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
        transl(0,-L3,0)*transl(L4,0,0);
    hold on
```

```
    x1=0;y1=0;z1=0;
    P1=[x1 y1 z1 1]';
    T1(:,i)=(Tbe)*P1;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    hold on;
    plot3(u1,v1,w1,'or'); %Dibujo vértices
end

for i=1:1:n_puntos(3) %Puntos de dibujo
    Puntos=Matriz_phantom(:,:,i)
    L3=Puntos(1)*Papel; %Coordenada x (30cm máximo)
    L4=Puntos(2)*Papel; %Coordenada y (m)

    angle=90;
    ROTACION=troty(angle,'deg');
    Tbe=transl(X,Y,Z)*troty(90,'deg')*ROTACION*
    transl(0,-L3,0)*transl(L4,0,0);

    %Definimos los puntos de nuestro minipanel%
    x1=0;x2=0;
    y1=0;y2=0;
    z1=-0.005;z2=-0.05; %Distancia seguridad

    P1=[x1 y1 z1 1]'; T1(:,i)=(Tbe)*P1;
    P2=[x2 y2 z2 1]'; T2(:,i)=(Tbe)*P2;

    u1=T1(1,i);v1=T1(2,i);w1=T1(3,i);
    Tp1(:,:,i)=[-1 0 0 u1;0 1 0 v1;0 0 -1 w1;0 0 0 1];

    u2=T2(1,i);v2=T2(2,i);w2=T2(3,i);
    Tp2(:,:,i)=[-1 0 0 u2;0 1 0 v2;0 0 -1 w2;0 0 0 1];

    Pintar(:,:,i)=[Tp1(:,:,i)];
    No_pintar(:,:,i)=[Tp2(:,:,i)];
end
```

- **Aplicacion_phantom_GUI.m**

En este archivo se leerán cada uno de los puntos dibujados en el panel para después enviar el conjunto de posiciones al robot simultáneamente.

```
n_lineas=size(Matriz_phantom);
posi=1;
orden=1;

for j=1:1:n_lineas(3)%Recorremos la Matriz_phantom
    orden=orden+1;
    a=0;
    if j==1 %Primera iteración
        posicion=Matriz_phantom(:,:,j);
        pos_img=j;

        if posicion(4)==1 %Pinta
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);
        elseif posicion(4)==0 %No pinta
            numero=No_pintar(:,:,pos_img);
        end
```

```
     else
        posicion=Matriz_phantom(:,:,j); %A partir de la segunda iteración
        posicion_anterior=Matriz_phantom(:,:,j-1);
        pos_img_ant=j-1;
        pos_img=j;

        if posicion(4)==1 & posicion_anterior(4)==1 %Dos puntos seguidos
            numero=Pintar(:,:,pos_img); %Dibuja Línea
        elseif posicion(4)==0
            numero=No_pintar(:,:,pos_img_ant);
        elseif posicion(4)==1 & posicion_anterior(4)==0 %Empezar a pintar
            numero(1:4,1:4)=No_pintar(:,:,pos_img);
            numero(1:4,5:8)=Pintar(:,:,pos_img);
        elseif posicion(3)==2
            numero=No_pintar(:,:,pos_img);
        end
    end

    for int=1:1:((length(numero)/4))
         Tpos=numero(1:4,1+a:4+a)
         pos=transl(Tpos); pos=pos*1000  %posiciones a milímetros
         rot=tr2rpy(Tpos); rot=rot*180/pi %ángulos a grados
         tcp(posi,:)=[rot(3) rot(2) rot(1) pos(1) pos(2) pos(3)]

         posi=posi+1;
         a=a+4;
    end

    if posi>=34 %34puntos máximo
        cont=1;
        for i=1:(posi-1) %Número de puntos
            tcps(cont:(cont+5))=tcp(i,:) %Coloca las tcp en una fila
            cont=(i*6)+1;
        end

        robot.TCPTrajectory(tcps)%Dibuja las líneas

        %Se mantiene a la espera hasta enviar siguiente trayectoria
        if orden<=n_lineas(3)
            robot.communicate
        else
            robot.socketfree;
        end

        posi=1;

        %Borrar en cada iteración
        tcps=[]; numero=[];%No todos tienen la misma dimensión
    end
end

%Cuando quedan menos de 34 puntos
if posi>1
    tcps=[];
    cont=1;
    for i=1:(posi-1) %Número de puntos de Tpos
        tcps(cont:(cont+5))=tcp(i,:) %Coloca las q en una fila
        cont=(i*6)+1;
    end
    robot.TCPTrajectory(tcps)%Dibuja los puntos
    robot.socketfree;
end
```

9. PLIEGO DE CONDICIONES

En este apartado se incluirán las características de las herramientas utilizadas.

9.1. Hardware

1. **Portátil *ASUS S551LN*:**
   - Procesador: *Intel® Core™ i7*
   - Memoria RAM: *8GB*
   - Tarjeta gráfica: *NVIDIA® GeForce 840M*

2. **Robot Industrial *ABB-IRB120*:**
   - Peso: 25kg
   - Altura: 580mm
   - Carga sportada: 3kg (hasta 4kg con la muñeca vertical)
   - Controlador: *IRC5 Compact*

3. **Dispositivo háptico PHANTOM® Omni**:
   - Máxima fuerza (brazos ortogonales): 3.3N
   - Fuerza nominal (esfuerzo continuado): 0.88N
   - Masa aparente del lápiz: 45g

9.2. Software

- Windows 10 © Microsoft Corporation.
- ABB RobotStudio
- MATLAB R2014a (con Toolbox de Robótica)
- Microsoft Office 365 ProPlus
- Librerías Haptik.

10. PRESUPUESTO

En este capítulo se proporciona información detallada sobre los costes teóricos del desarrollo del proyecto, incluyendo los costes de materiales y las tasas profesionales.

10.1. Costes Materiales

Objeto		Cantidad	Precio Unidad [€]	Precio Total [€]
Hardware	Portátil ASUS	1	999,00	999,00
	ABB-IRB120	1	10954,00	10954,00
	PHANTOM Omni	1	1000,00	1000,00
Software	Microsoft Windows 10	1	0,00[8]	0,00
	RobotStudio	1	0,00	0,00
	MATLAB R2014a	1	0,00	0,00
	Microsoft Office 365 ProPlus	1	0,00	0,00
	Librerías Haptik	1	0,00[9]	0,00
TOTAL				**12953,00**

TABLA 10.1 Costes materiales (hardware y software) sin IVA.

[8] Los precios por unidad son gratuitos debido a que son versiones de software gratis para estudiantes.
[9] Es gratuito debido a que es software libre.

10.2. Tasas Profesionales

Las tasas profesionales incluyen los costes de realización del proyecto correspondientes al tiempo dedicado a su desarrollo.

Objeto	Tiempo [meses]	Coste unidad [€/mes]	Precio Total [€]
Ingeniería	4	1350	5400
Escritura	1	650	650
TOTAL			**6050,00**

10.3. Costes Totales

Los costes totales del proyecto han sido obtenidos sumando los costes materiales y profesionales aplicando el IVA.

Objeto	Costes Totales
Costes materiales	12953,00 €
Costes profesionales	6050,00 €
Subtotal	19003,00€
IVA (21%)	3990,63€
TOTAL	**22993,63€**

TABLA 10.2 Costes totales con IVA

11. MANUAL DE USUARIO

En esta sección se detallará el funcionamiento del sistema a través de la interfaz de usuario desarrollada, exponiendo las posibilidades y funcionalidades de cada elemento de interacción que conforma las aplicaciones. Además, se incluirá una guía de botones para un mejor entendimiento del funcionamiento.

11.1. Arranque del sistema

La interfaz GUI está desarrollada para que al ejecutar la aplicación principal o escribir el nombre del archivo aparezca una ventana emergente con la interfaz lista para utilizarse.

Para poder utilizar posteriormente la aplicación del Phantom, será necesario que se establezca como fichero principal la carpeta de la librería Haptik, la cual contiene las funciones necesarias para la inicialización del dispositivo, y que los archivos desarrollados en MATLAB para las aplicaciones se añadan a su ubicación.

Para iniciar la interfaz se escribirán las siguientes líneas de código[10]:

```
>> inicializaTFG
>> Interfaztfg
```

[10] Asegurarse que el archivo *inicializaTFG.m* se ejecuta una única vez debido a que en caso contrario aparecerá un error interno en MATLAB y se tendrá que cerrar.

La ejecución del primer archivo nos permite la creación de variables globales en el *Workspace* (espacio de trabajo) de MATLAB, de modo que en todo momento podemos tener acceso a los datos que guardan; mientras que el segundo archivo ejecuta la interfaz general directamente:

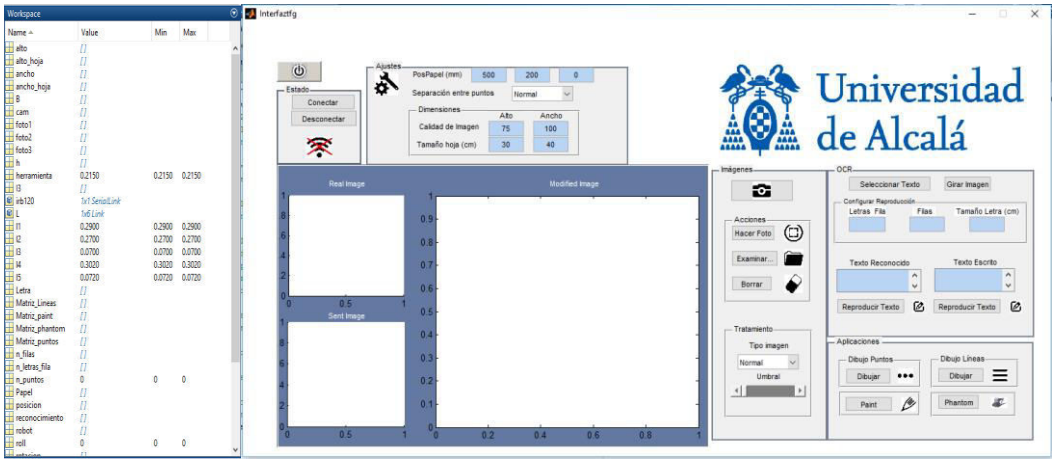


FIGURA 11.1 Workspace e interfaz general en MATLAB

Para poder utilizar las aplicaciones, será necesario primero establecer la conexión con el robot, para ello tendremos que ejecutar el sistema robot, donde tenemos dos posibilidades que serán descritas en los siguientes puntos.

11.1.1. Conexión vía simulación

Para la creación del socket de comunicación mediante RAPID se ha creado el archivo *RS_PositionControlSocket_Bidirectional.rspag,* que contiene todas las funciones necesarias, y que tendremos que ejecutar mediante el programa RobotStudio. Una vez que hayamos desempaquetado la estación del robot nos encontraremos dentro del programa, que mostrará la siguiente apariencia:

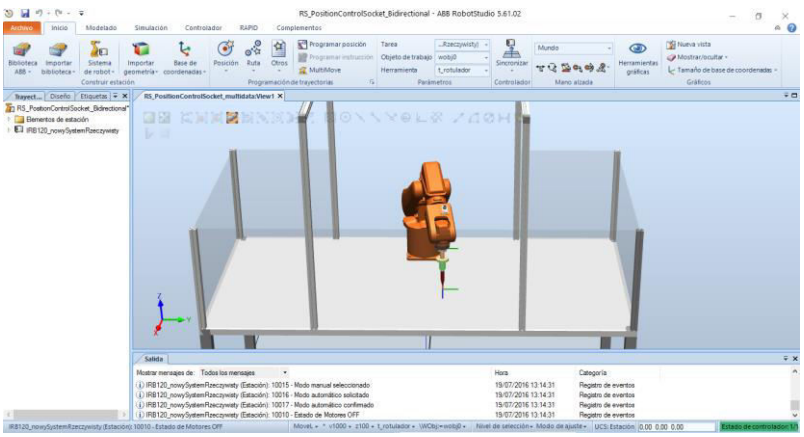


FIGURA 11.2 Estación de trabajo RobotStudio

Lo primero que haremos será comprobar si la dirección IP es la correcta para establecer la comunicación, para verificarlo seguiremos la siguiente secuencia:

> Pestaña *RAPID* → Pestaña *Controlador* → Desplegable *RAPID* → →Programa *T_ROB1*→ Módulo *TCPIPConnectionHandler*

Donde encontraremos la siguiente línea de código:

```
23          ipAddress := "127.0.0.1";
```

Si queremos trabajar en simulación en el mismo equipo que estamos ejecutando MATLAB, la dirección IP deberá ser 127.0.0.1, mientras que, si queremos trabajar en conexión de red con diferentes computadoras, primero deberemos comprobar la dirección de la misma.

Todo esto puede realizarse sencillamente en un ordenador con sistema operativo Windows, buscando *Símbolo del Sistema* y escribiendo en la ventana de comandos *ipconfig*:

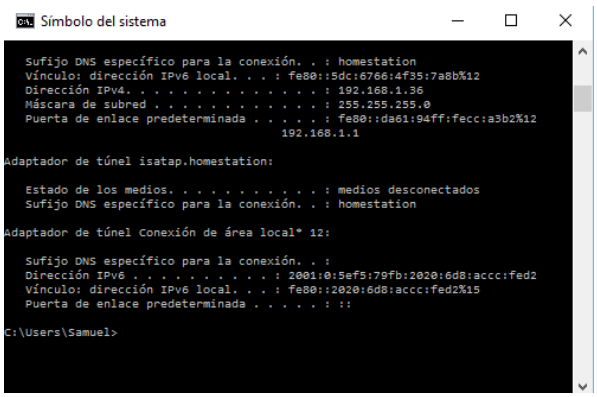


FIGURA 11.3 Ventana de símbolo del sistema

La dirección IP correspondiente a red de conexión se encuentra en la línea *Dirección IPv4*, y debemos introducirla en la línea correspondiente del programa en RobotStudio, y de la interfaz en MATLAB para que coincidan:

```
687 -           robot=irb120('127.0.0.1',1024);
```

Una vez esté establecida la dirección IP con la que se va a trabajar, en la pestaña *Simulacion* en RobotStudio, se pulsará en *Monitor,* y activaremos el rastreo del TCP, que va a pintar las trayectorias realizadas por el robot y va a permitir vislumbrar los resultados obtenidos. Para garantizar que pueda dibujarse toda la imagen se recomienda establecer la máxima longitud de rastreo permitida y la utilización de un color oscuro.

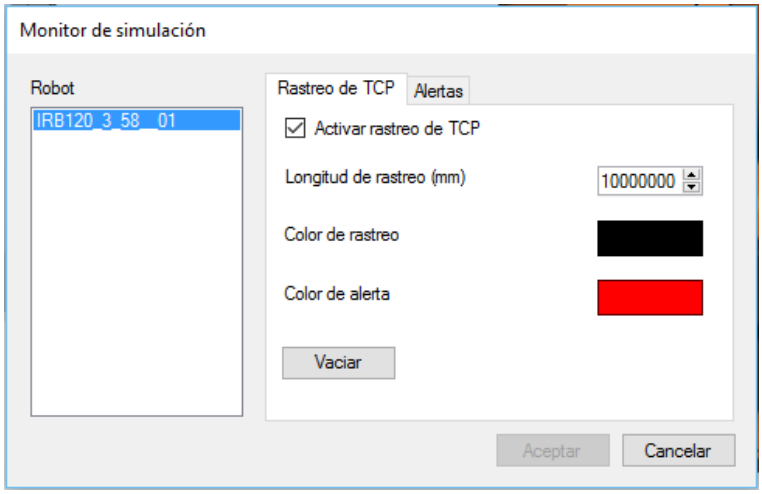


FIGURA M.11.4 Monitor de simulación en RobotStudio

Tras esto, sobre la misma pestaña pulsaremos en *Reproducir*, y nos mantendremos a la espera de que el usuario inicie la conexión por MATLAB.

11.1.2. Conexión vía brazo robot IRB120

El proceso previo a la conexión requiere abrir el simulador de RobotStudio y comprobar del mismo modo que la dirección IP es 172.29.28.185, que es la que se corresponde con el controlador del robot real. Después se guardará el programa *T_ROB1* (realizando *click derecho* sobre su nombre) en una unidad de almacenamiento externo.

Para arrancar el robot debemos actuar sobre las llaves de la siguiente imagen. Girar *Power switch* a *ON* y seleccionar el modo de funcionamiento (*Mode switch*) en automático (izquierda) o manual (derecha).

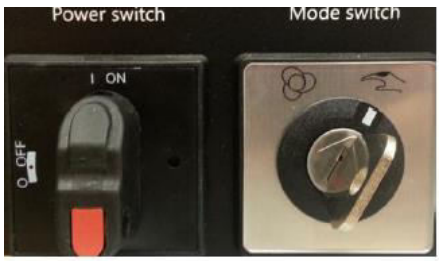


FIGURA 11.5 Llaves de funcionamiento del controlador IRC5

Para realizar un control seguro se aconseja utilizar el modo manual que requiere que se mantenga pulsado durante la ejecución del robot un botón situado en el lateral del *Flexpendant* como se observa en la siguiente imagen.

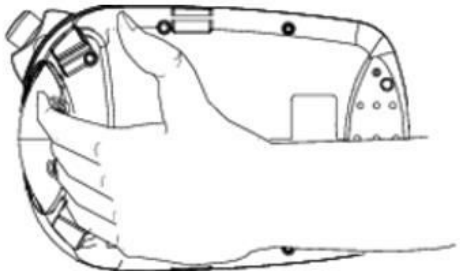


FIGURA 11.6 Botón de ejecución brazo robot en modo manual

Una vez ha arrancado el sistema se encenderá la pantalla del *FlexPendant* donde accediendo a *Datos del Programa* debemos buscar el programa guardado en la unidad de almacenamiento externo que previamente se debe haber conectado en la entrada del dispositivo. Después se cargará el programa y en *Ventana de Producción* tendremos el programa desarrollado en RAPID.

Antes de arrancar el sistema debemos asegurarnos de que el robot se encuentra con la herramienta rotulador como efector final, ya que, por seguridad, es aconsejable realizar su manipulación antes de que pueda moverse.

Para iniciar la ejecución debe pulsarse en *PP a main,* que situará el puntero del programa al inicio del mismo, y a continuación pulsaremos el botón físico de *play* del FlexPendant.

FIGURA 11.7 Ventana de navegación del FlexPendant

Una vez que se está ejecutando el sistema robot, ya sea en simulación o en el brazo robot, realizamos la conexión mediante la interfaz en MATLAB. En el recuadro *Estado* pulsaremos sobre *Conectar* y nos aparecerá una ventana que nos preguntará el tipo de conexión que queremos realizar.

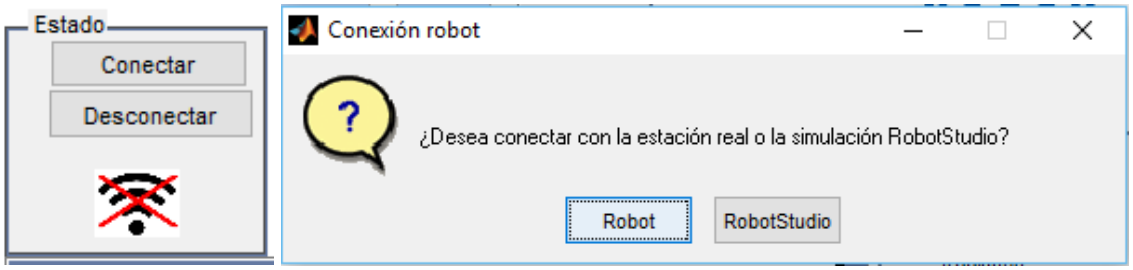


FIGURA 11.8 Inicio conexión de MATLAB con el robot

Tras haber elegido una de las opciones, aparecerá una barra de carga que mostrará un mensaje cuando se complete y desaparecerá la cruz roja que se encuentra encima del icono de conexión.

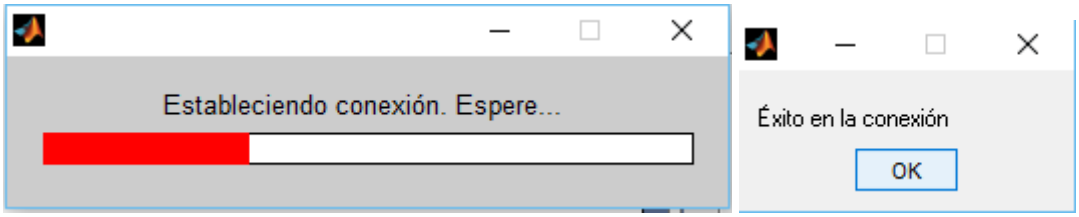


FIGURA 11.9 Diálogo de conexión

11.2. Utilización de la interfaz

11.2.1. Ajustes de reproducción

Este apartado es común a todas las aplicaciones y es el primer paso que debe realizarse antes de ejecutar cualquier aplicación, aunque siempre puede modificarse antes de efectuar un envío.

El panel empleado para este proceso se corresponde con el de *Ajustes,* el cual podemos ver en la siguiente imagen:

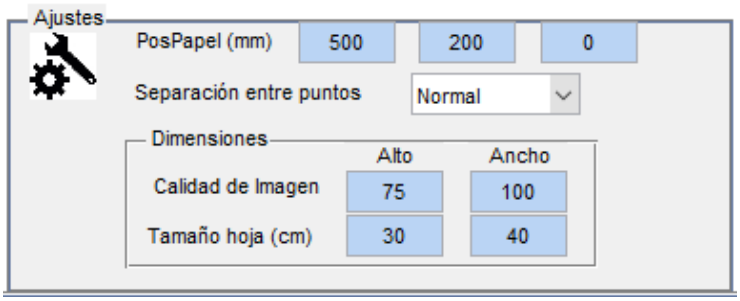


FIGURA 11.10 Panel de ajustes de la interfaz gráfica

Como podemos comprobar, inicialmente vienen establecidos unos valores que se corresponden a la configuración normal para una hoja de tamaño DIN-A3 colocada en posición horizontal.

Los recuadros correspondientes a *PosPapel* indican la posición en milímetros en la que se coloca la esquina superior izquierda del papel sobre el que se va a dibujar, correspondiendo a la posición en *X, Y, Z*, respectivamente, siendo la posición de origen la de la base del robot.

En el recuadro *Dimensiones*, vamos a poder establecer el tamaño de la hoja que se va a utilizar, definiendo su alto y su ancho, que modificará automáticamente los valores de *Calidad de Imagen*, para obtener la mayor resolución posible en el dibujo de imágenes.

Finalmente, la opción *Separación de puntos*, se emplea para la reproducción de imágenes mediante líneas o puntos y permite la elección de una separación normal, pequeña o grande, que va a determinar el máximo número de píxeles en el alto y ancho de la imagen y, por tanto, modifica también la *Calidad de Imagen*.

11.2.2. Ejecución de las aplicaciones

En este apartado se va a describir el procedimiento que se debe seguir para efectuar un correcto funcionamiento de cada una de las aplicaciones.

a) Reconocimiento y escritura de textos

Esta aplicación consiste en la reproducción del texto contenido en una imagen o que ha sido escrito mediante el teclado.

Para poder adquirir una imagen se utilizará el recuadro de *Acciones,* que se encuentra dentro del panel *Imágenes* en la interfaz, que contiene las opciones que podemos ver a continuación:

FIGURA 11.11 Panel de adquisición de imágenes para reconocimiento de texto

Con el primer botón se abrirá una ventana que mostrará lo que esté visualizando la cámara que esté conectada al ordenador, y en *Hacer Foto*, podrá realizarse una instantánea.

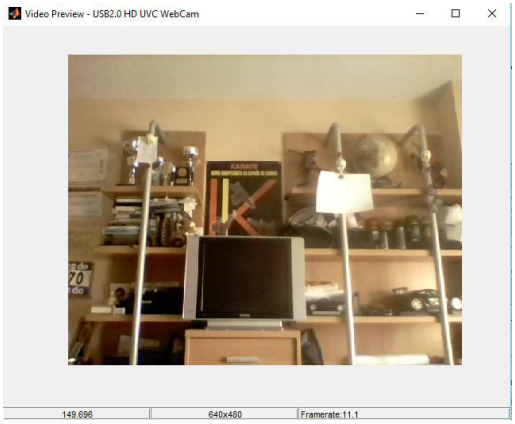


FIGURA 11.12 Visualización de cámara conectada al ordenador

Con el botón de *Examinar...* se abrirán una ventana que permite explorar las carpetas dentro del mismo equipo para seleccionar una imagen.

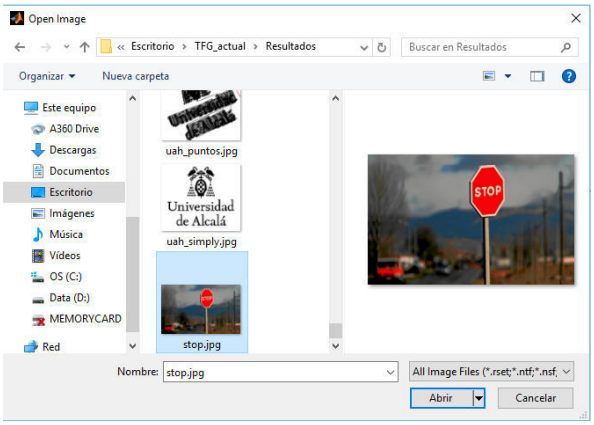


FIGURA 11.13 Adquisición de imágenes del propio equipo

Si la imagen seleccionada resulta no ser la adecuada, puede borrarse para obtener otra mediante el botón *Borrar.*

Tras seleccionar la imagen se empleará el panel *OCR:*

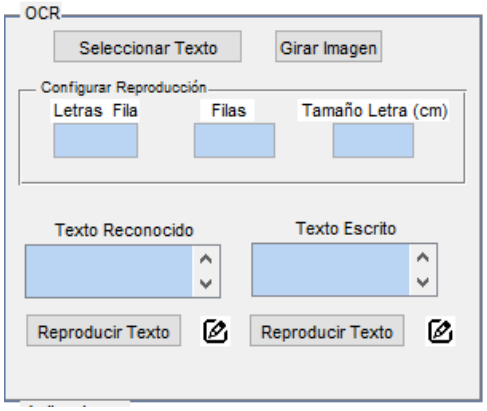


FIGURA 11.14 Panel para la reproducción de textos

Si el texto no se encuentra totalmente recto, seleccionaremos la opción *Girar Imagen*, donde tendremos que encuadrar la zona de texto que queremos reconocer, realizando *click* sobre *Modified Image* en el siguiente orden:

Esquina superior izquierda → Esquina superior derecha → Esquina inferior izquierda → Esquina inferior derecha → ENTER

Una vez que el texto de la imagen se muestre completamente recto, pulsaremos sobre *Seleccionar Texto*, donde tendremos que recuadrar la zona que queremos reconocer manteniendo pulsado el ratón hasta conseguirlo.

FIGURA 11.15 Ejemplo proceso de reconocimiento de texto

Después de esto, en el recuadro de *Texto Reconocido* aparecerá el texto que buscamos si el reconocimiento se ha realizado adecuadamente.

El siguiente paso es el envío de la información al robot, para ello previamente definiremos el valor de los recuadros *Letras Fila, Filas* y *Tamaño Letra*, en función del texto que queramos enviar. A continuación, pulsaremos sobre *Reproducir Texto* y el robot comenzará a escribir.

El proceso de adquisición del texto escrito se produce al introducirlo vía teclado en el recuadro *Texto Escrito,* el resto de procedimiento para el envío es el mismo.

b) Reproducción de imágenes mediante puntos y líneas

A pesar de ser aplicaciones diferentes, el proceso para su utilización es el mismo, por lo que se explicará conjuntamente.

En primer lugar, será necesario adquirir la imagen que se desea reproducir. Esta función se encuentra dentro del panel *Imágenes:*

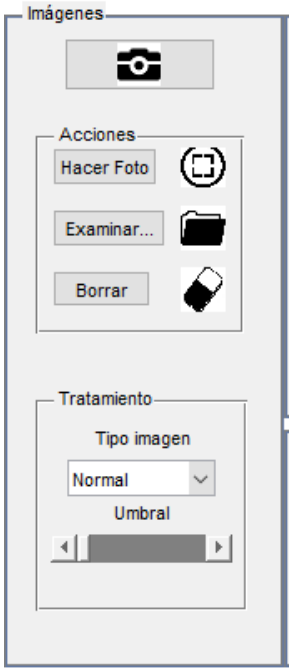


FIGURA 11.16 Panel de adquisición y transformación de imágenes

El proceso para la adquisición es idéntico al de reconocimiento de textos, por lo que no se va a incidir más en ello.

Una vez que tengamos adquirida la imagen, será necesario que le apliquemos un tratamiento para facilitar su representación en el papel, ya que únicamente se dispone de un color para pintar.

En el panel *Tratamiento* encontraremos el desplegable *Tipo imagen* donde podemos encontrar 3 tipos de tratamientos:

- *Normal*: realiza una umbralización (convierte la imagen a blanco y negro) de la imagen en función del valor de umbral escogido en la barra deslizable *Umbral*.
- *Dilatación*: realiza una umbralización y a continuación una dilatación que permite que se incluyan píxeles de dibujo en la imagen que han podido perderse, de nuevo en función del valor de umbral escogido en la barra deslizable *Umbral*.
- *Erosión:* realiza una umbralización y, a continuación, una erosión que permite eliminar información innecesaria de la imagen quedándose con la parte esencial, en función del valor de umbral escogido en la barra deslizable *Umbral*.

Para la visualización del proceso de transformación vamos a contar con 3 imágenes:

- *Real Image*: muestra la imagen tal cual se adquiere.
- *Modified Image*: muestra la imagen tras aplicar el tratamiento escogido.
- *Sent Image*: muestra la imagen de píxeles que va a ser enviada al robot.

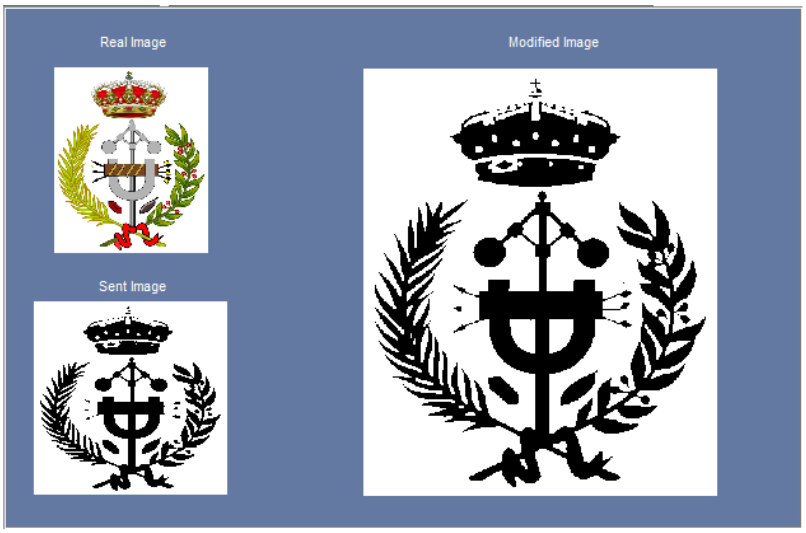


FIGURA M.11.17 Imagen adquirida modificada

Una vez que la imagen ha sido modificada por el usuario, se realiza el envío de la información desde el panel *Aplicaciones,* para hacer un dibujo por puntos o líneas.

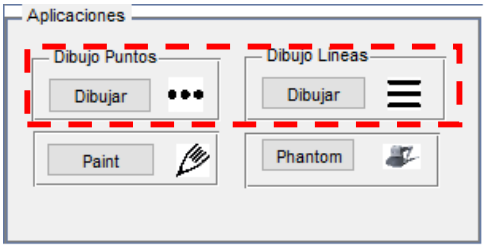


FIGURA 11.18 Panel de aplicaciones para dibujo puntos y líneas

c) Aplicación tipo Pizarra

Para acceder a esta aplicación debe pulsarse sobre el panel *Aplicaciones* en el botón *Paint:*

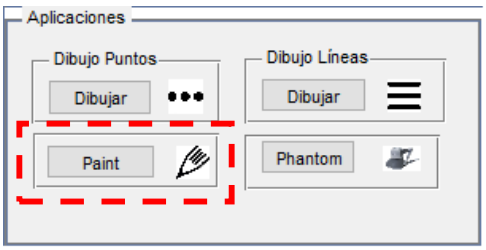


FIGURA 11.19 Panel de aplicaciones para dibujo tipo pizarra

Tras esto se abrirá en una nueva ventana una interfaz como la siguiente:

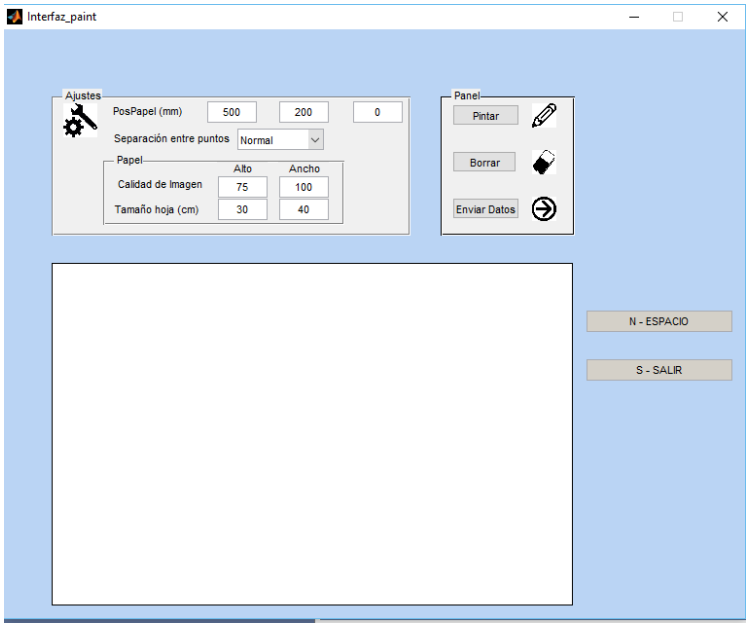


FIGURA 11.20 Ventana interfaz aplicación tipo pizarra

En esta nueva interfaz encontraremos el panel *Ajustes* con las mismas opciones que habíamos visto para la interfaz general, por lo que no se entrará en más detalle.

Para iniciar la adquisición de imágenes pulsaremos sobre *Pintar* y realizaremos *click* sobre el recuadro blanco donde queramos realizar un punto. Si volvemos a *clickar* en otro punto se trazará una línea recta, si se pulsa la tecla *N* del teclado servirá para finalizar la última línea dibujada y dar comienzo a la siguiente en otro punto. Para finalizar el dibujo, pulsaremos sobre la tecla *S.*

En caso de que se quiera realizar un dibujo nuevo se podrá eliminar el actual pulsando sobre *Borrar*. En caso contrario, puede enviarse la información al robot en *Enviar* para que comience a dibujar.

d) Aplicación Phantom

Del mismo modo que antes, para acceder a esta aplicación debe pulsarse sobre el panel *Aplicaciones* en el botón *Phantom*:

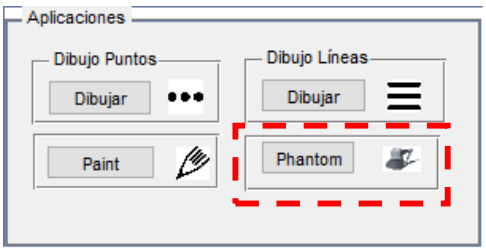


FIGURA 11.21 Panel de aplicaciones para dibujo con Phantom

A continuación, se abrirá en una nueva ventana una interfaz como la siguiente:

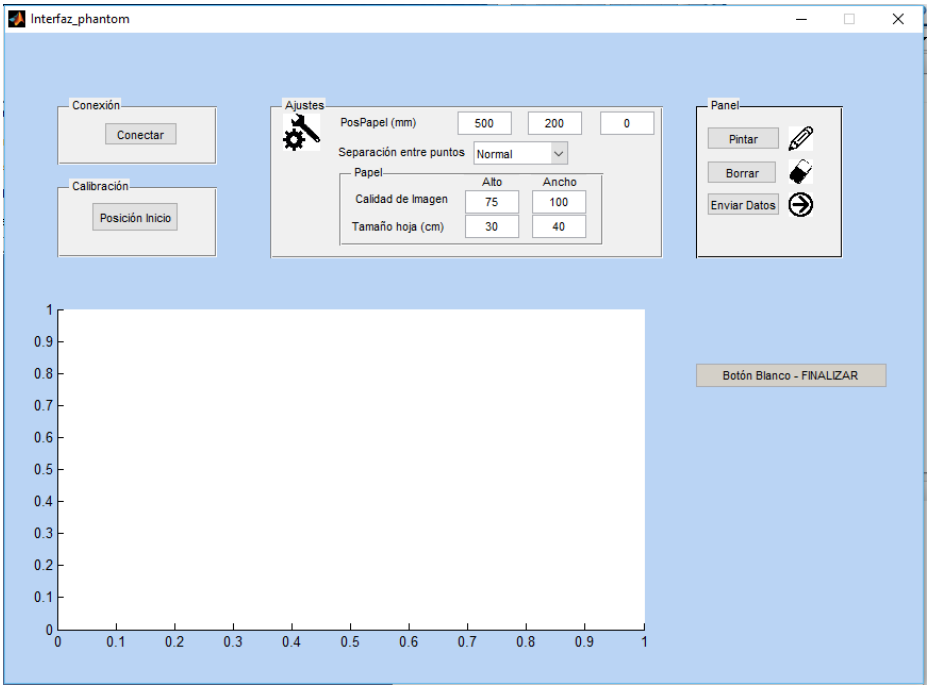


FIGURA 11.22 Ventana interfaz aplicación Phantom

De nuevo encontramos en esta interfaz el panel de *Ajustes* que tiene la misma funcionalidad explicada anteriormente.

Para poder ejecutar esta aplicación lo primero que tendremos que hacer es realizar la conexión del Phantom. Para ello pulsaremos sobre el botón *Conectar*[11] y aparecerán por pantalla las siguientes líneas de código:

[11] Es posible que la primera vez que se realiza la conexión, MATLAB se quede cargando durante mucho tiempo, en ese caso se recomienda cerrar el programa y volverlo a abrir.

```
ans =

    [0]    'Default PHANToM'    'Omni'             'SensAble Technologies'
    [1]    'Mouse Spectre 0'    'Mouse Spectre'    'Siena Robotics and...'

>> h=haptikdevice

h =

    haptikdevice object: 1-by-1
```

Por defecto quedará seleccionado el dispositivo que se encuentre conectado. En el momento que se haya establecido la conexión se encenderá una luz azul en el Phantom como observamos en la siguiente imagen, lo que indicará que podemos comenzar su ejecución.

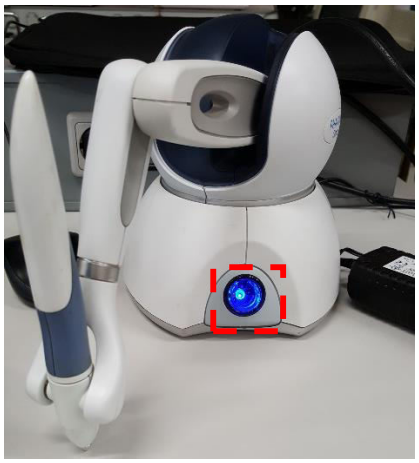


FIGURA 11.23 Dispositivo Phantom conectado

Tras esto, lo que tenemos que hacer es configurar la posición de inicio para realizar la calibración. Se realizará pulsando *Posicion Inicio,* que mantendrá el programa a la espera de que se pulse el botón gris del Phantom. La posición en la que debe colocarse el lápiz antes de pulsar el botón debe ser la esquina izquierda superior del área donde va a realizarse el dibujo.

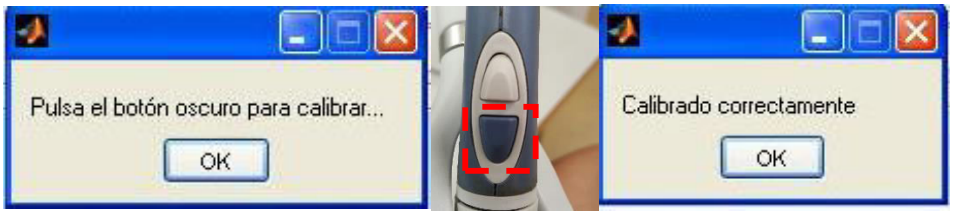


FIGURA 11.24 Calibración posición Phantom

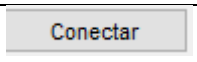
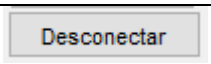
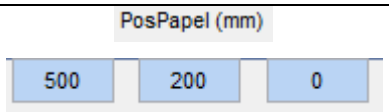
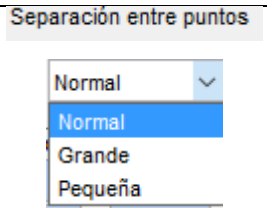
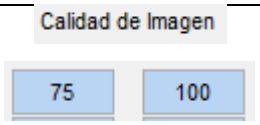
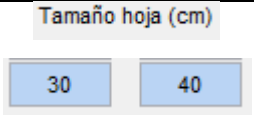
Una vez que se encuentra calibrado, podemos comenzar a dibujar pulsando en el botón *Pintar*, y a continuación, pulsando el botón gris del lápiz del Phantom. El dibujo se representará en el recuadro blanco de la interfaz al mismo tiempo que movemos el lápiz

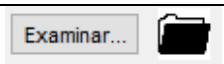
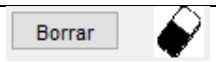
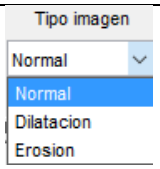
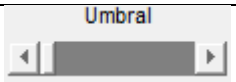
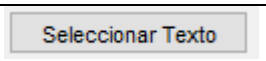
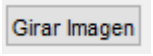
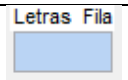
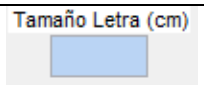
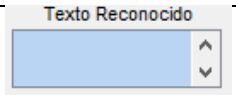
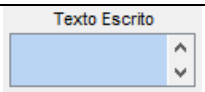
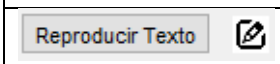
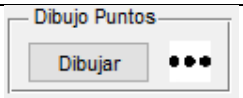
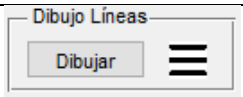
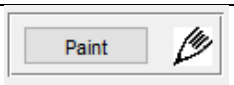
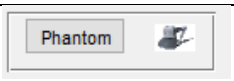
sobre el área establecida para el dibujo. Debe tenerse en cuenta que se dibujará cuando el lápiz esté apoyado en el papel, mientras que dejará de dibujar cuando lo levantemos.

Para finalizar el dibujo debemos presionar el botón blanco del lápiz. Si el resultado obtenido es el deseado podemos enviar la información al robot clickando sobre *Enviar Datos*, en caso contrario podemos borrarlo pulsando en *Borrar* y comenzar un nuevo dibujo del mismo modo.

11.2.3. Resumen de la funcionalidad de los elementos de la interfaz

En este apartado va a hacer un resumen sobre todos los botones contenidos en la interfaz junto con su función.

Interfaz	Panel	Control	Función
General	—		Lleva al robot a la posición de reposo tras terminar de realizar una aplicación.
	Estado		Conecta MATLAB con la simulación o el robot real, según la elección del usuario.
			Desconecta MATLAB de la simulación o del robot real, según su conexión.
General, Paint y Phantom	Ajustes		Determina la posición de la esquina superior izquierda (X, Y y Z respectivamente) donde se coloca el papel sobre el que se va a dibujar respecto a la base del robot.
			Establece la separación entre los puntos de dibujo en el papel, una separación *Grande* hará que los puntos consecutivos se encuentren más separados, mientras que una separación *Pequeña*, que estén más juntos.
			Indica el número de puntos que contiene la imagen (alto y ancho respectivamente). Se modificará en función del tamaño del papel y la separación de puntos.
			Establece las dimensiones del papel sobre el que se va a dibujar (alto y ancho, respectivamente)

Interfaz	Panel	Control	Función
General	Imágenes		Abre una ventana que permite visualizar en tiempo real la cámara conectada al equipo.
			Permite tomar una instantánea de lo que se encuentra visualizando la cámara.
			Accede a los archivos del ordenador para cargar una imagen.
			Borra una imagen adquirida por la cámara o desde el equipo.
			Sirve para establecer el tratamiento que se quiere dar a la imagen adquirida para su posterior representación mediante puntos o líneas.
			Cuantifica mediante una barra deslizable el tratamiento escogido en el control anterior.
	OCR		Selecciona mediante un recuadro el área de una imagen sobre la que se quiere adquirir el texto.
			Realiza el giro de una imagen para poder adquirir el texto contenido correctamente.
			Establece el número de letras representadas por el robot en cada fila de dibujo.
			Determina el número de filas sobre las que va a escribir el robot.
			Establece el tamaño de cada letra que va a ser escrita.
			Muestra en el recuadro el texto reconocido de la imagen adquirida.
			Permite escribir en el recuadro el texto que posteriormente sea enviado al robot.
			Envía al robot el texto reconocido o escrito para su representación.
	Aplicaciones		Transfiere al robot la orden para reproducir la imagen mediante puntos.
			Envía al robot la orden para reproducir la imagen mediante líneas.
			Abre la interfaz de la aplicación tipo pizarra.
			Abre la interfaz de la aplicación Phantom.

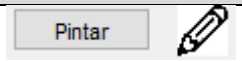
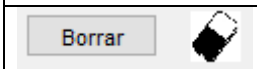
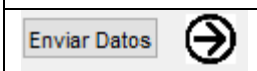
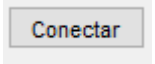
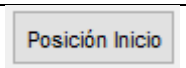
Interfaz	Panel	Control	Función
Paint y Phantom	Panel		Inicia la adquisición del dibujo representado sobre el cuadro blanco.
			Permite borrar la imagen dibujada una vez terminada.
			Envía la información de la imagen dibujada al robot.
Phantom	Conexión		Establece la conexión del Phantom con el ordenador.
	Calibración		Permite la calibración de la posición leída por el dispositivo colocándolo en la esquina superior izquierda del área de dibujo.

TABLA 11.1 Resumen de los controles de la interfaz

BIBLIOGRAFÍA

[1] G. Patiño González, "Reproducción de imágenes y textos con el robot IRB120", Trabajo de Fin de Grado, Universidad de Alcalá de Henares, 2015.

[2] A. Gutiérrez Corbacho, "Desarrollo de una interfaz para el control del robot IRB120 desde MATLAB", Trabajo de Fin de Grado, Universidad de Alcalá de Henares, 2014.

[3] A. Barrientos, L.F. Peñín, C. Balaguer, R. Aracil, *Fundamentos de Robótica,* 2ª ed. Madrid: McGraw-Hill, 1997.

[4] M.J. Frydrysiak, "Socket Based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station", Trabajo de Fin de Grado.

[5] Página Web Oficial Productos ABB. [Online]. Disponible: http://new.abb.com/products/robotics/es

[6] Manual de referencia técnica. *Instrucciones, funciones y tipos de datos de RAPID,* ABB Robotics, Zúrich, Suiza, 2010.

[7] Página Web Oficial Productos Geomagic. [Online]. Disponible: http://www.geomagic.com/es/products/phantom-omni/overview/

[8] Librería Haptik. [Online]. Disponible: http://sirslab.dii.unisi.it/haptiklibrary/

[9] M.L. Pinto Salamanca, "Análisis e Implementación de una Interfaz Háptica en Entornos Virtuales", Tesis Magíster, Universidad Nacional de Colombia, 2009.

[10] *Haptik Library 1.0,* Siena Robotics and Systems Lab., Siena, Italia, 2005.

[11] Página Web Oficial Productos MATLAB. [Online]. Disponible: http://es.mathworks.com/products/

[12] P.I. Corke, Toolbox Robótica. [Online]. Disponible: http://petercorke.com/Robotics_Toolbox.html

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá